计算机学院专业必修课

计算机组成

流水线及其冒险

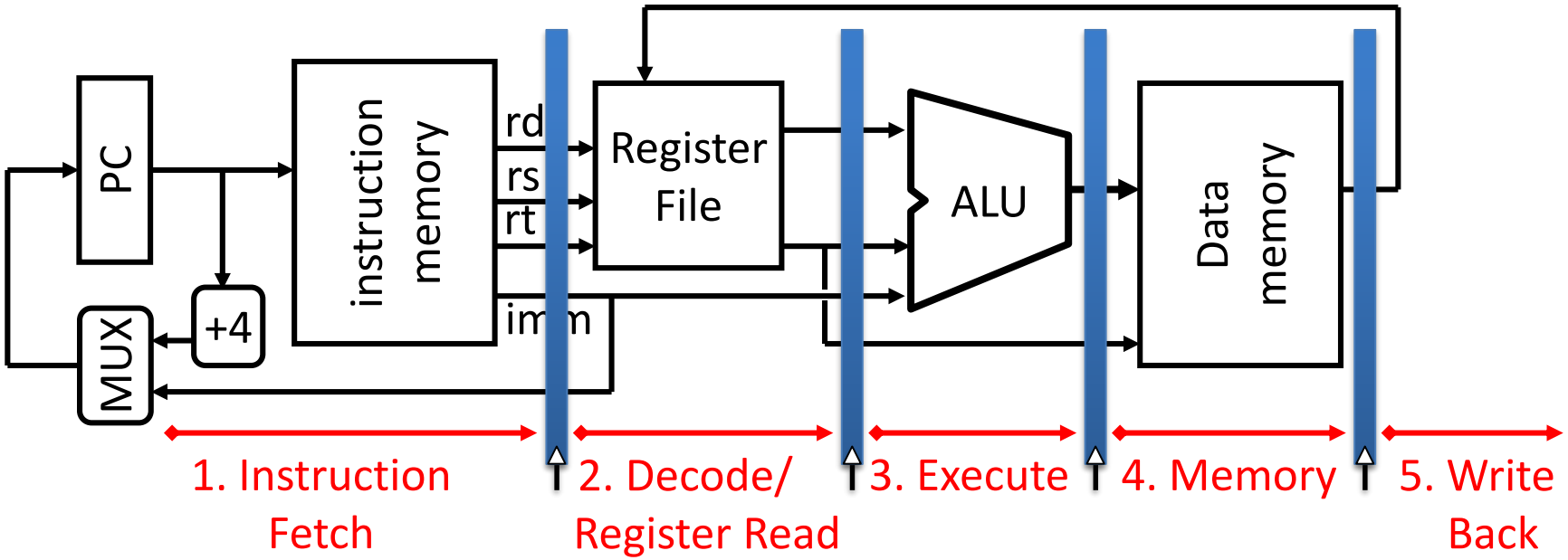
高小鹏

北京航空航天大学计算机学院

Recall: 5 Stages of MIPS Datapath

1) IF: Instruction Fetch, Increment PC
2) ID: Instruction Decode, Read Registers
3) EX: Execution (ALU)
   Load/Store: Calculate Address
   Others: Perform Operation
4) MEM:
   Load: Read Data from Memory
   Store: Write Data to Memory
5) WB: Write Data Back to Register

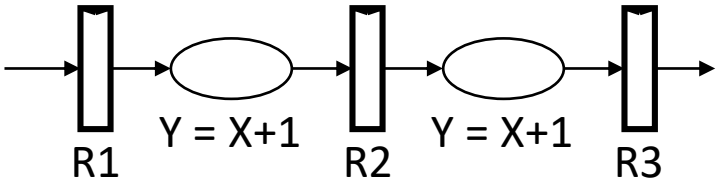
Pipelined Datapath

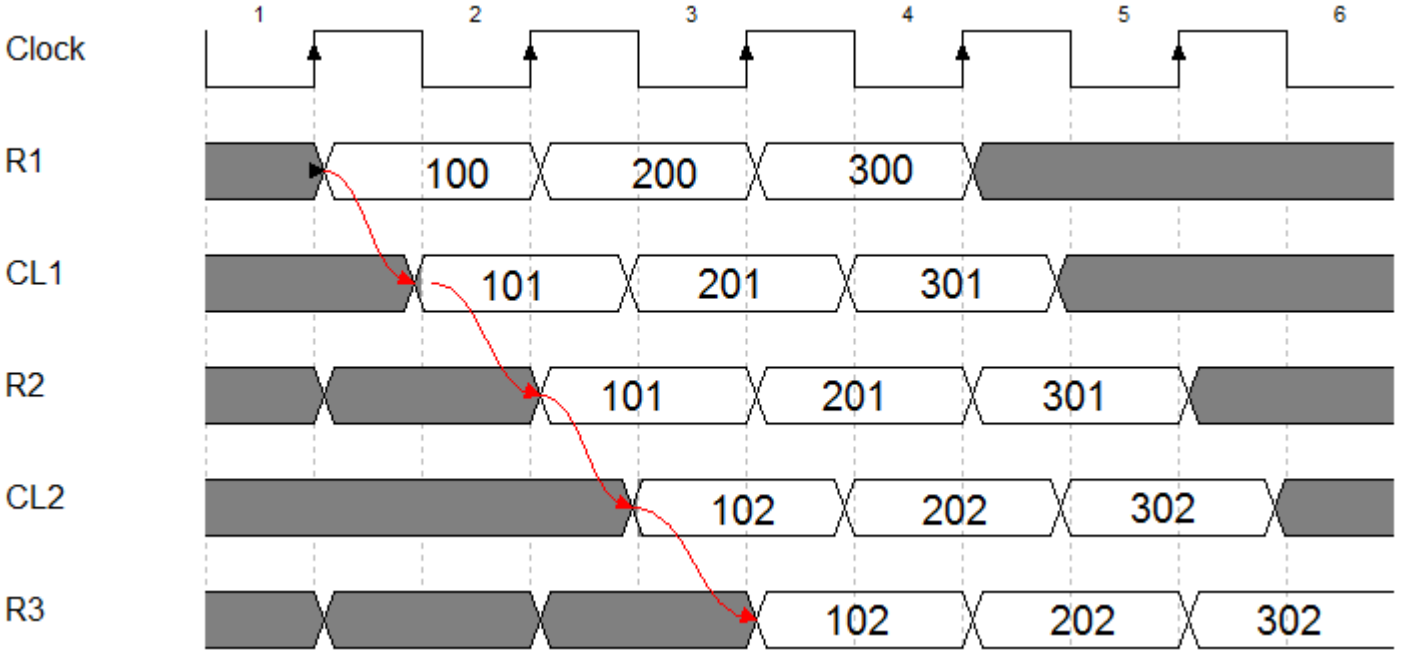


- Add registers between stages
  - Hold information produced in previous cycle
- 5 stage pipeline
  - Clock rate *potentially* 5x faster

流水线的本质

- 示例：一般数字电路的流水线特性

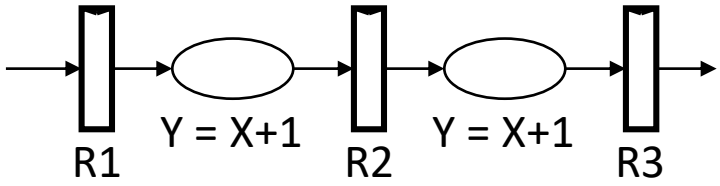




- 通过流水寄存器，物理切割指令执行的5个步骤
- 由于寄存器分割了组合逻辑，因此多条指令能同时执行
  - 不同指令位于不同流水段，即：不同流水段的组合逻辑服务于不同的指令

流水线的电路工作过程的形式表示

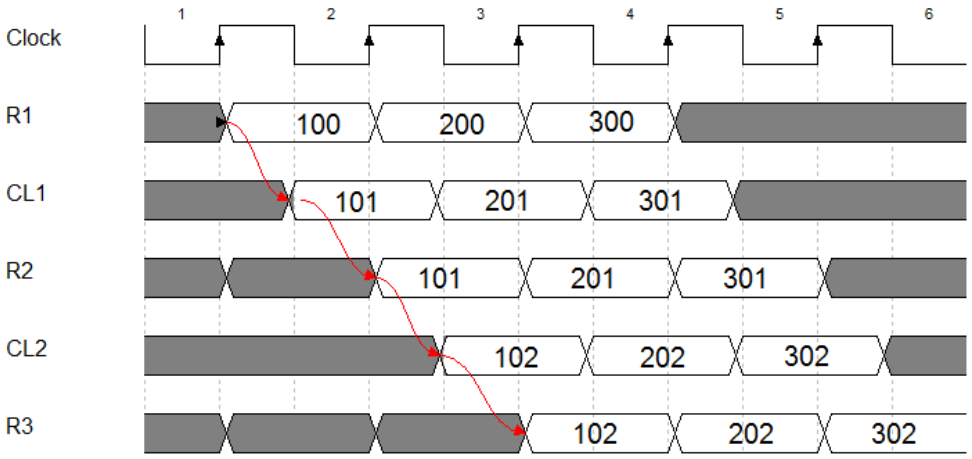
- 工作过程采用表格表示
  - 水平方向：寄存器、组合逻辑
  - 垂直方向：时钟



- 推演方法

$$R_i^N = F_i(R_{i-1}^{N-1})$$

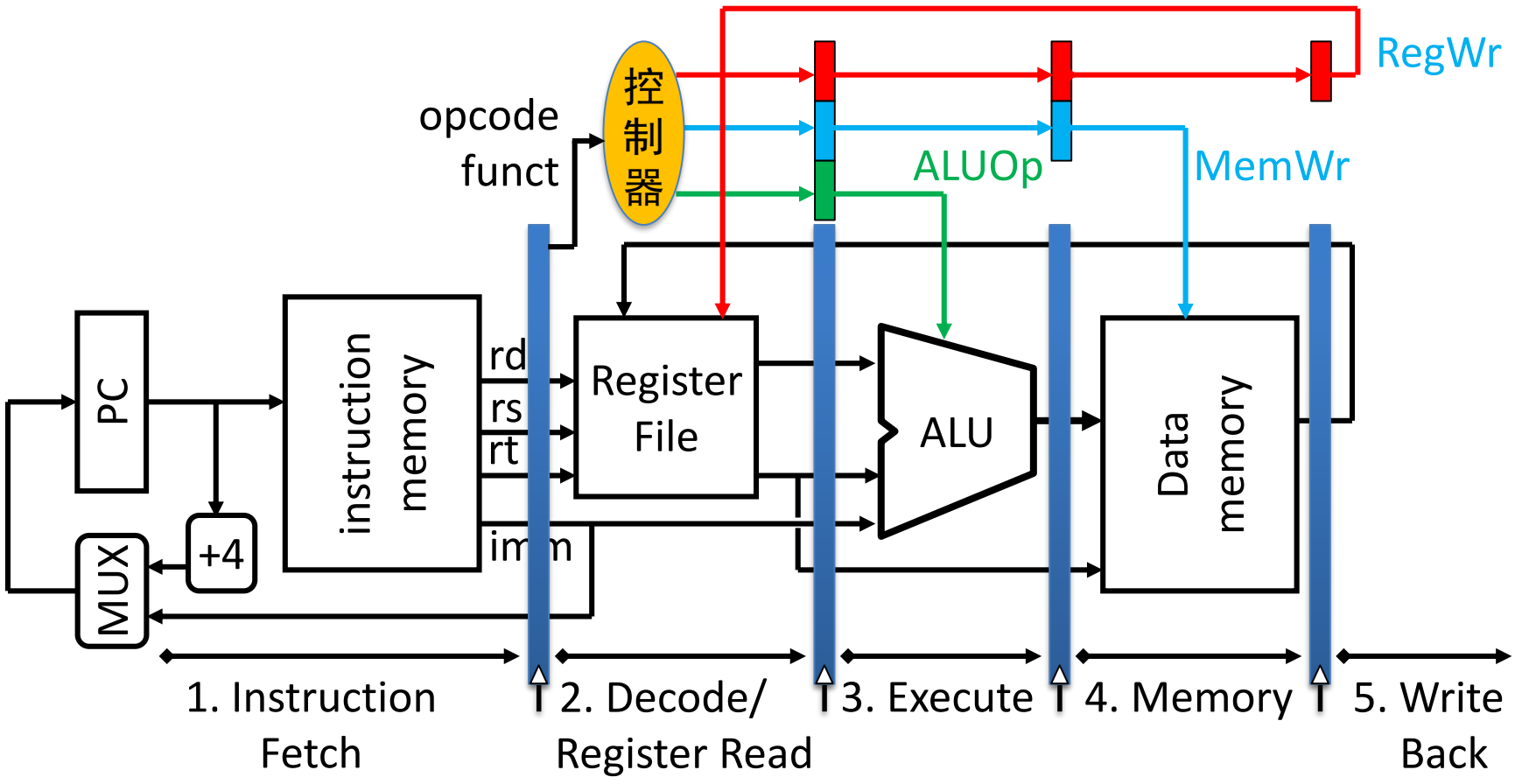
  - 1、以时钟周期为单位推演
  - 2、各级R的取值：由前级R在前一周期时的值及组合逻辑决定



	CL1	CL2	
CLK	R1	R2	R3
↑ 1	XXX→100		
↑ 2	100→200	101	
↑ 3	200→300	201	102
↑ 4		301	202
↑ 5			302

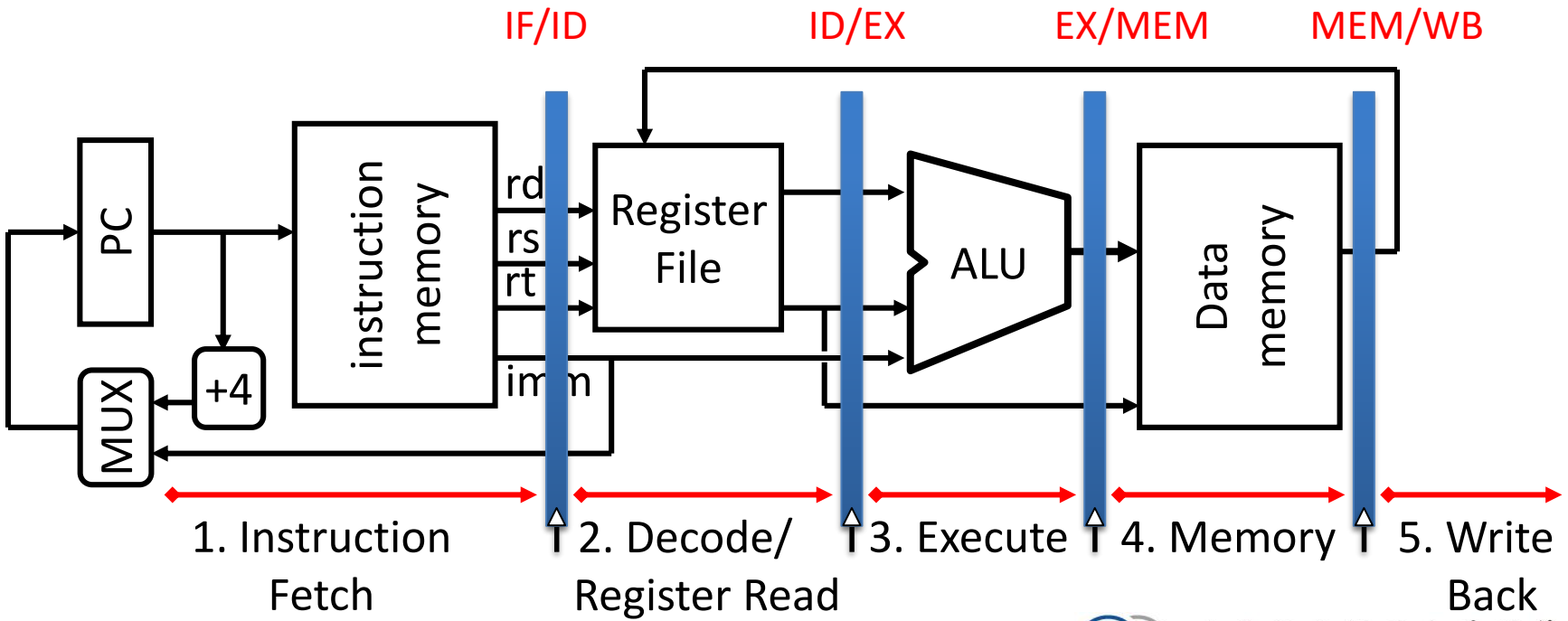
流水的控制信号

- 控制器：译码产生控制信号，与单周期完全相同
- 控制信号流水寄存器：控制信号在寄存器中传递，直至不再需要
  - 注意：控制信号 = 指令！



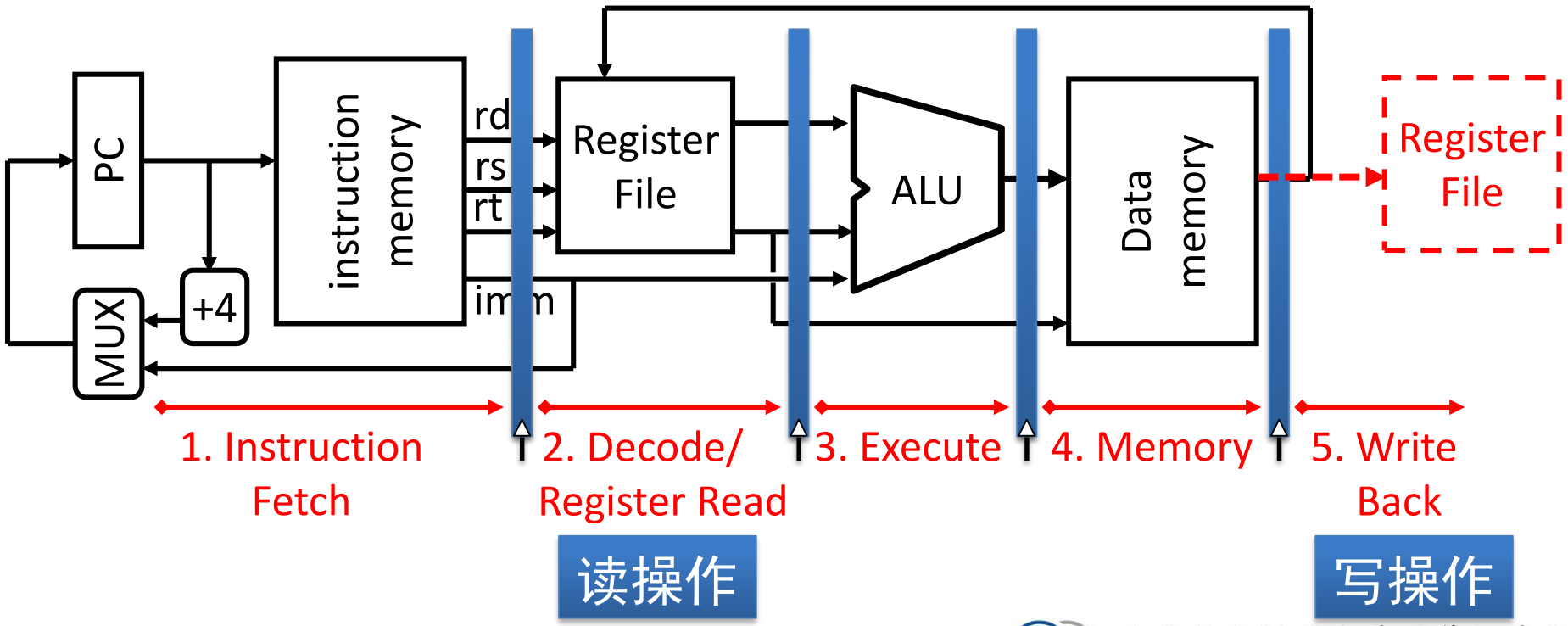
正确认识流水线—流水线寄存器

- 命名法则：前级/后级
  - 示例：IF/ID，前级为读取指令，后级为指令译码(及读操作数)
- 功能：时钟上升沿到来时，保存前级结果；之后输出至下级组合逻辑
  - 也可能直接连接到下级流水线寄存器
    - 例如ID/EX保存的从RF读出的第2个寄存器值，就直接传递到EX/MEM

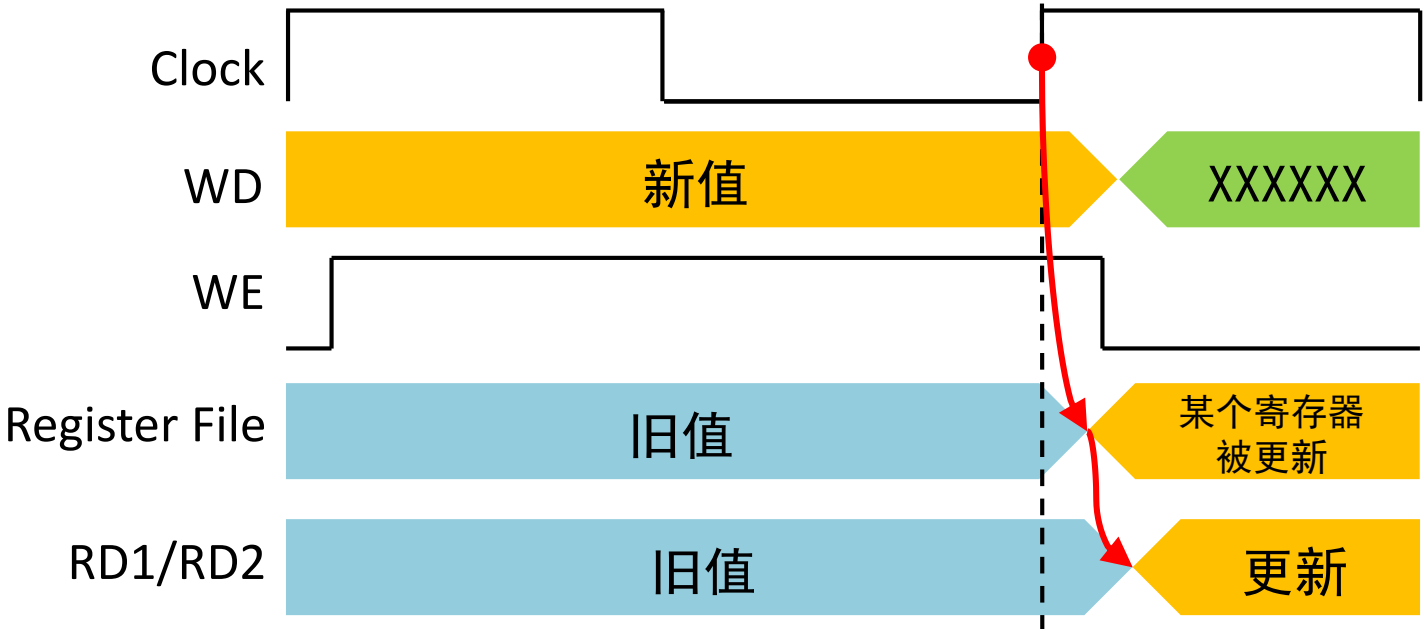


正确认识流水线：流水线级数与RF

- N级流水线：必须有N级流水线寄存器
  - 插入N-1级流水线寄存器，最后一级为Register File
  - RF：ID阶段为读出操作；WB阶段为写入操作
    - 读出：组合逻辑行为；写入：寄存器行为

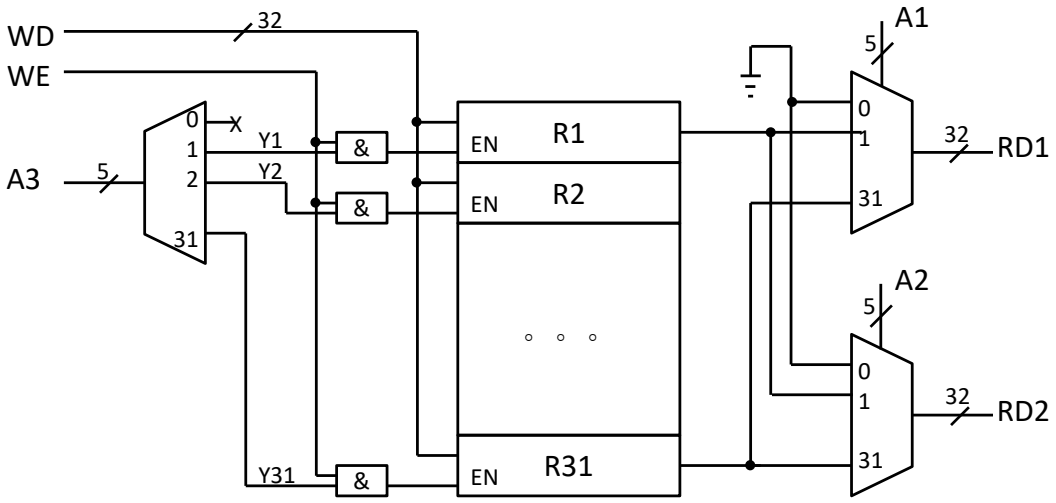


正确认识流水线：流水线级数与RF



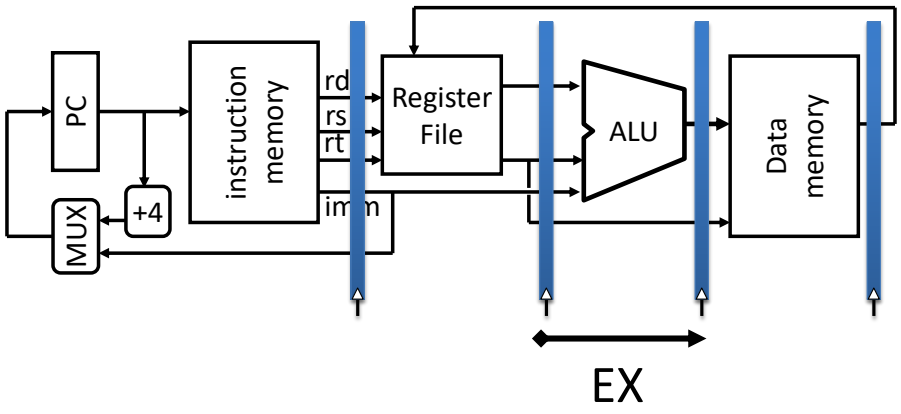
- RF具有2重行为
  - 读出：组合逻辑
  - 写入：时序逻辑(寄存器)

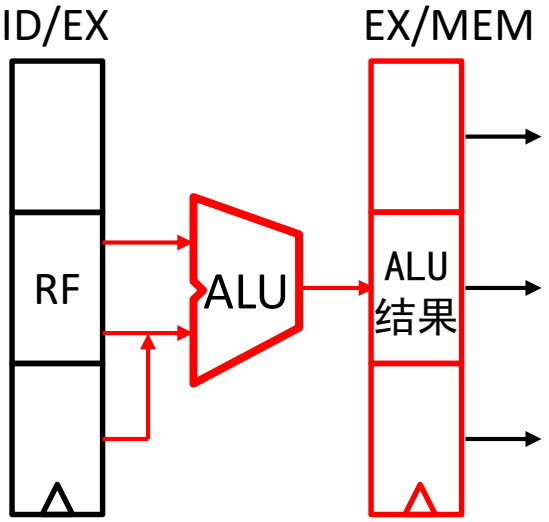
- 写入：上升沿写入
  - WE需有效
- 输出：与时钟无关，仅与RF内容及A1/A2相关



正确认识流水线：流水段与流水线寄存器

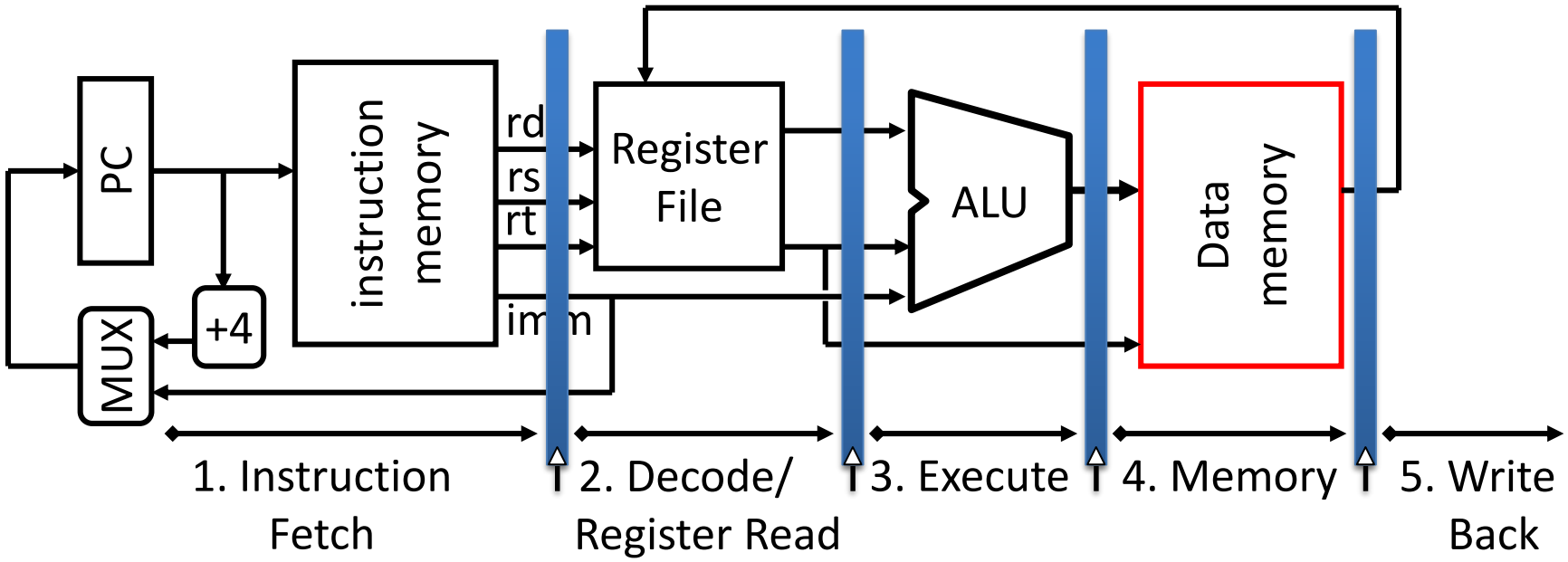
- 流水段：组合逻辑+寄存器
  - 起始：前级流水线寄存器的输出
  - 中间：组合逻辑（如ALU）
  - 结束：写入后级流水线寄存器
  - 当时钟上升沿到来时，组合逻辑计算结果存入后级寄存器
- 示例：EX阶段
  - 起始：ID/EX的2个寄存器值及立即数扩展值的输出
  - 中间(组合逻辑)：ALU完成计算
  - 结束(寄存器)：在clock上升沿到来时，结果写入EX/MEM中相应寄存器





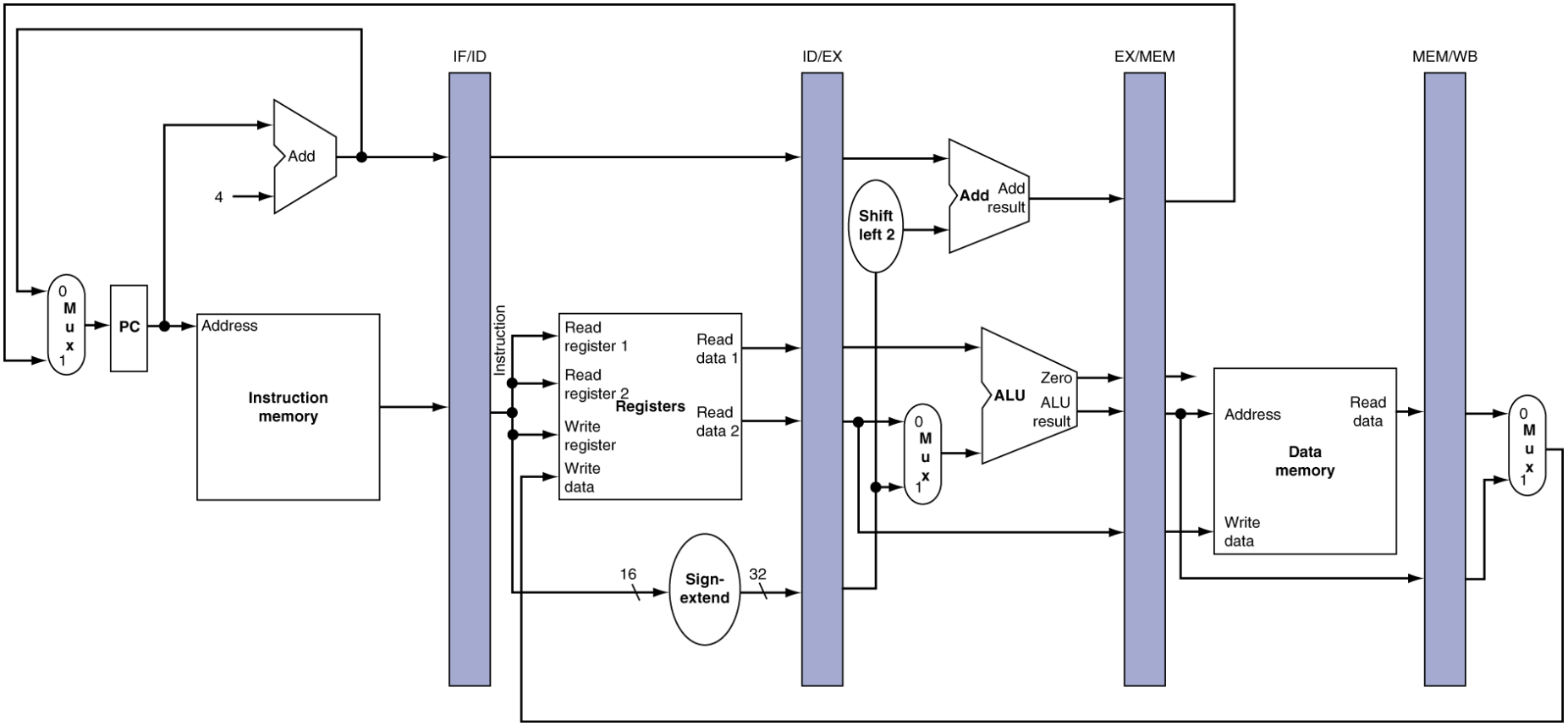
正确认识流水线：DM

- 写入时：表现为寄存器。在时间上与MEM/WB寄存器同级
- 读出时：可以等价为组合逻辑
  - 与RF的读出是类似的

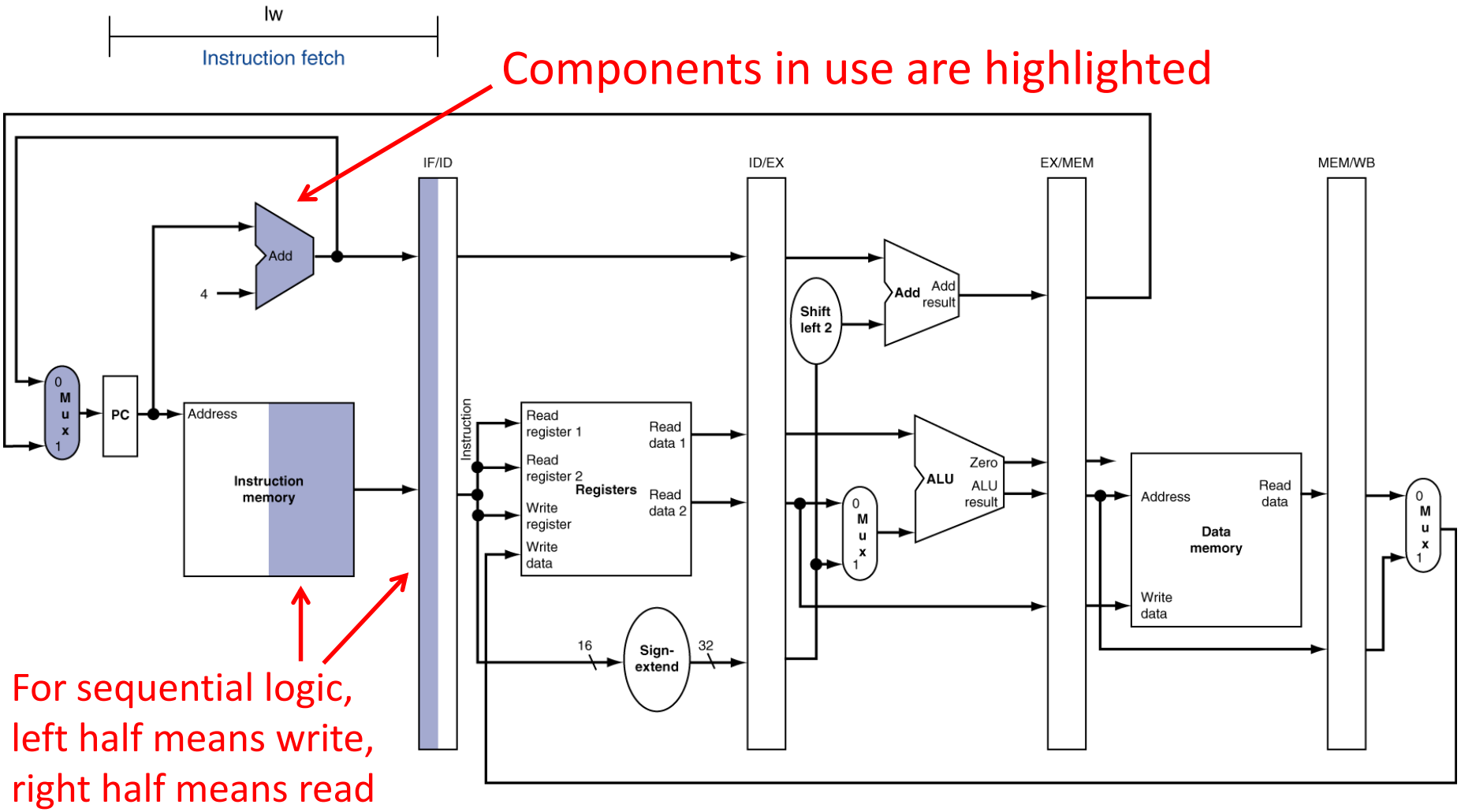


More Detailed Pipeline

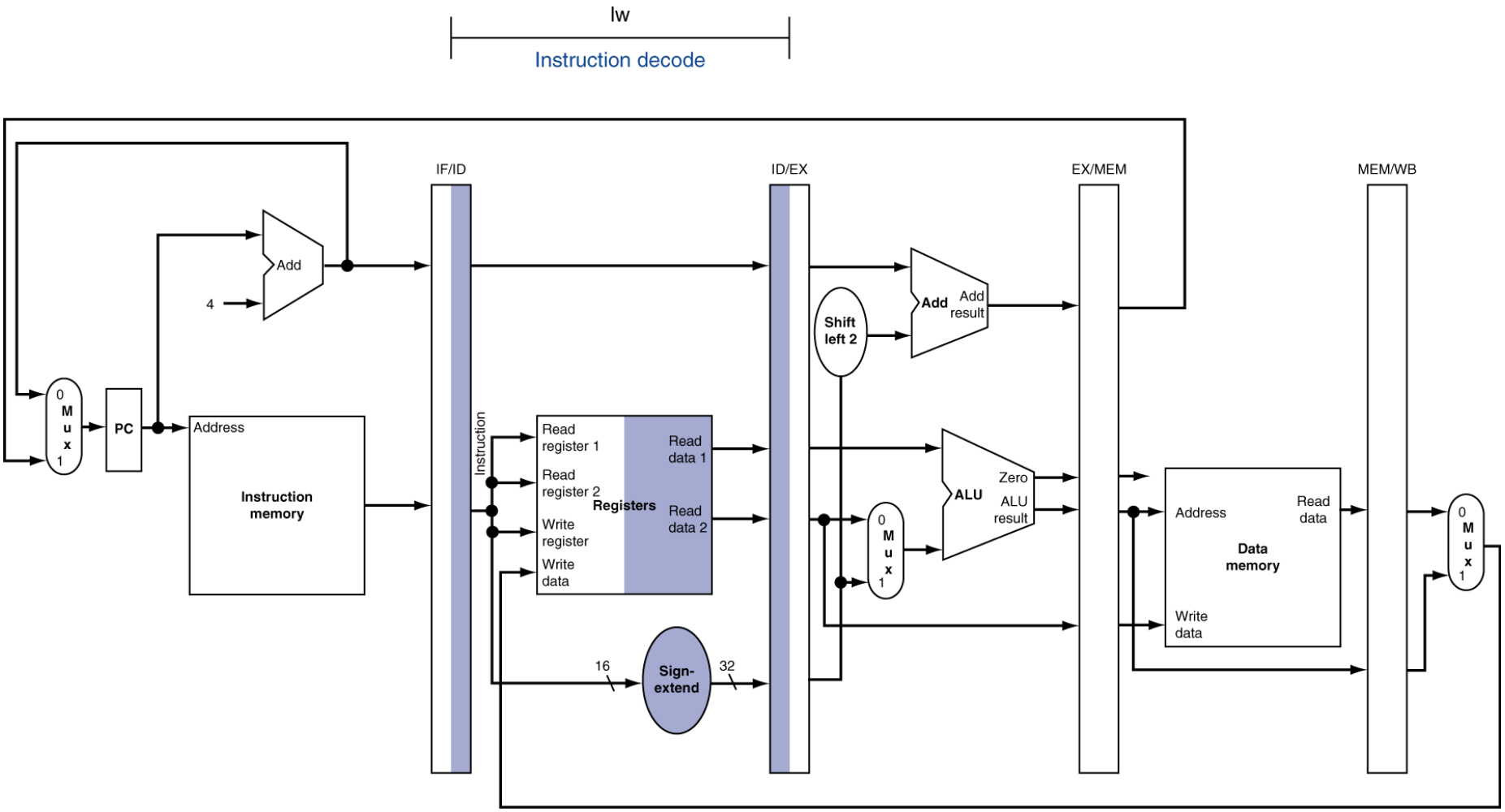
- Examine flow through pipeline for `lw`



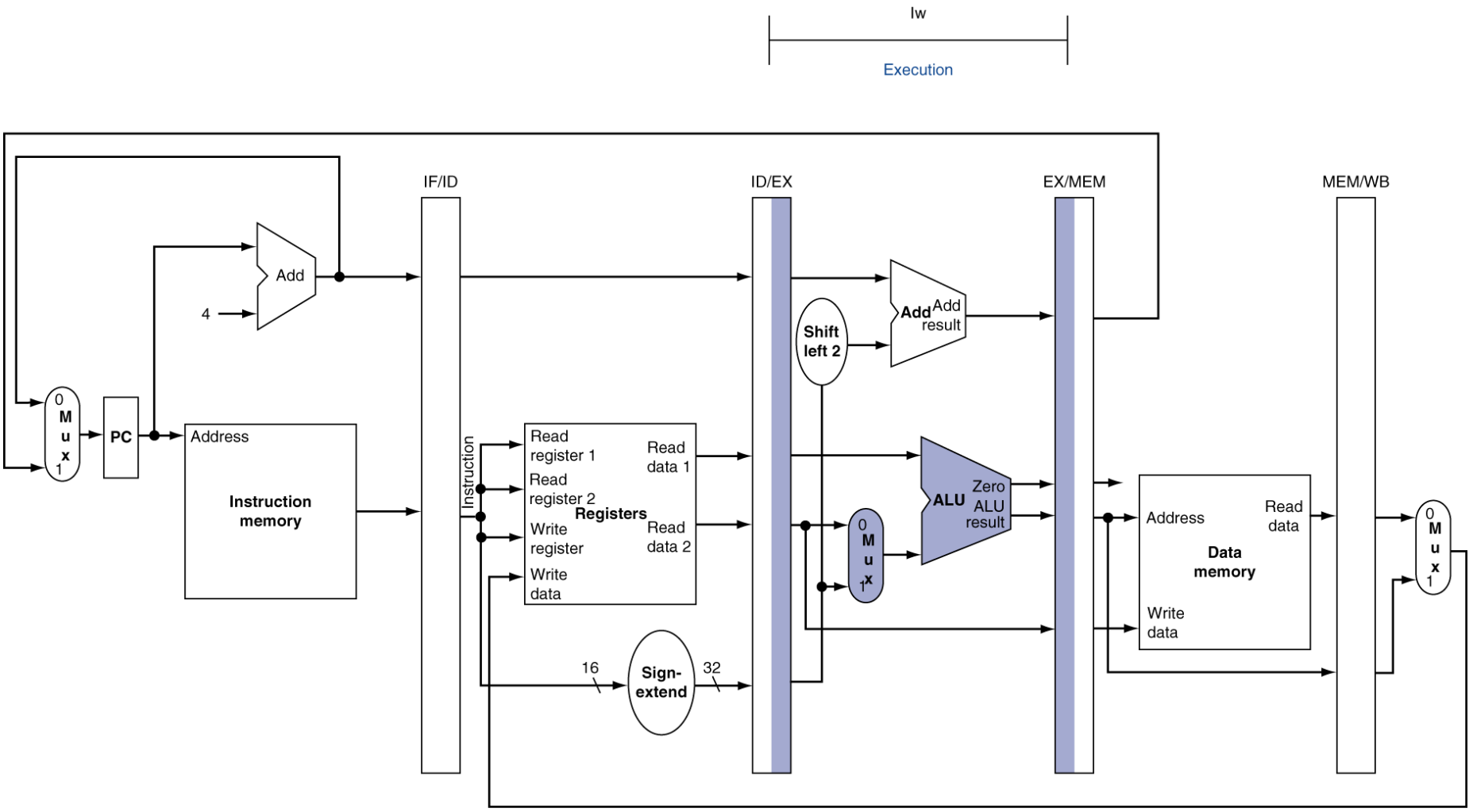
Instruction Fetch (IF) for Load



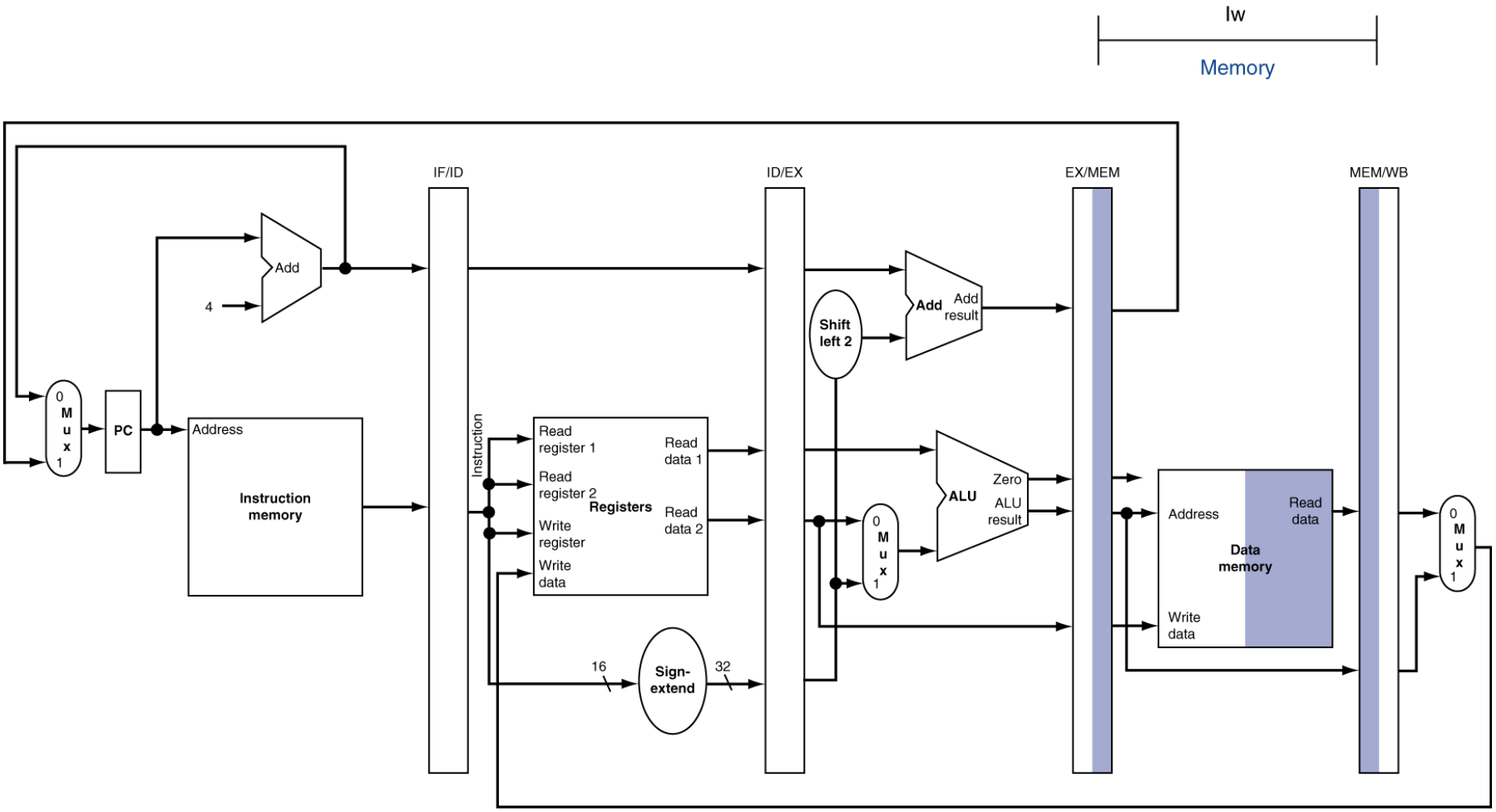
Instruction Decode (ID) for Load



Execute (EX) for Load

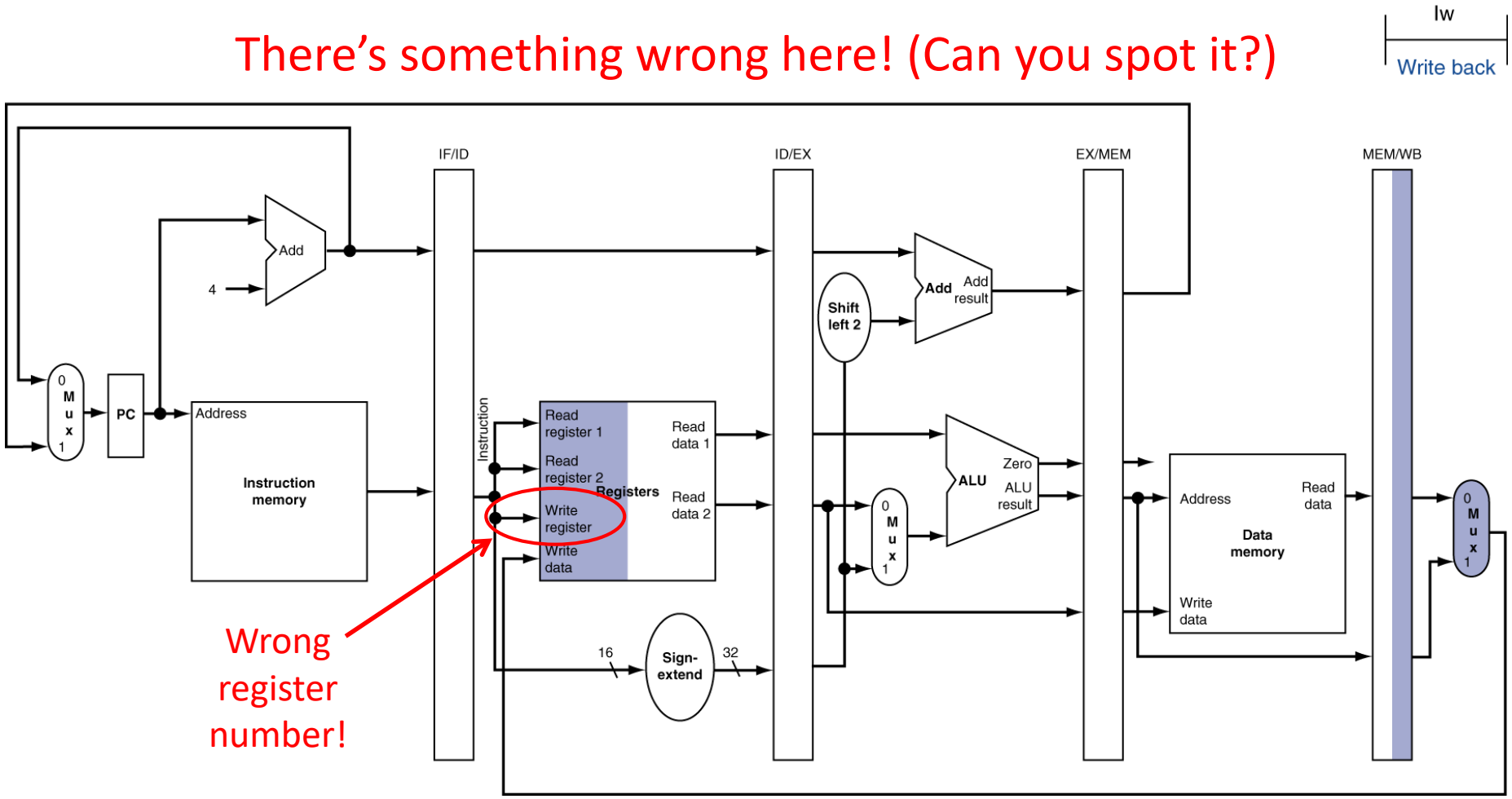


Memory (MEM) for Load



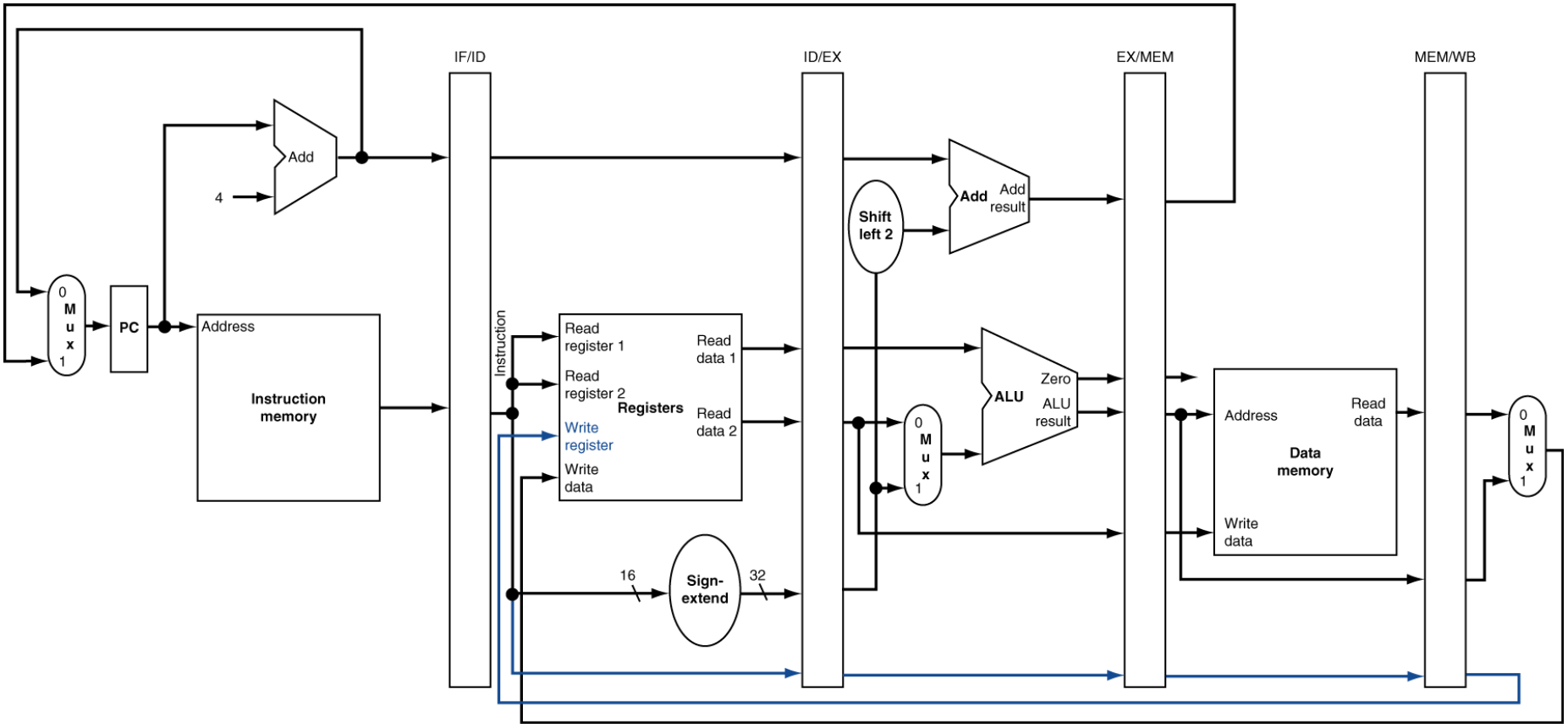
Write Back (WB) for Load

There's something wrong here! (Can you spot it?)

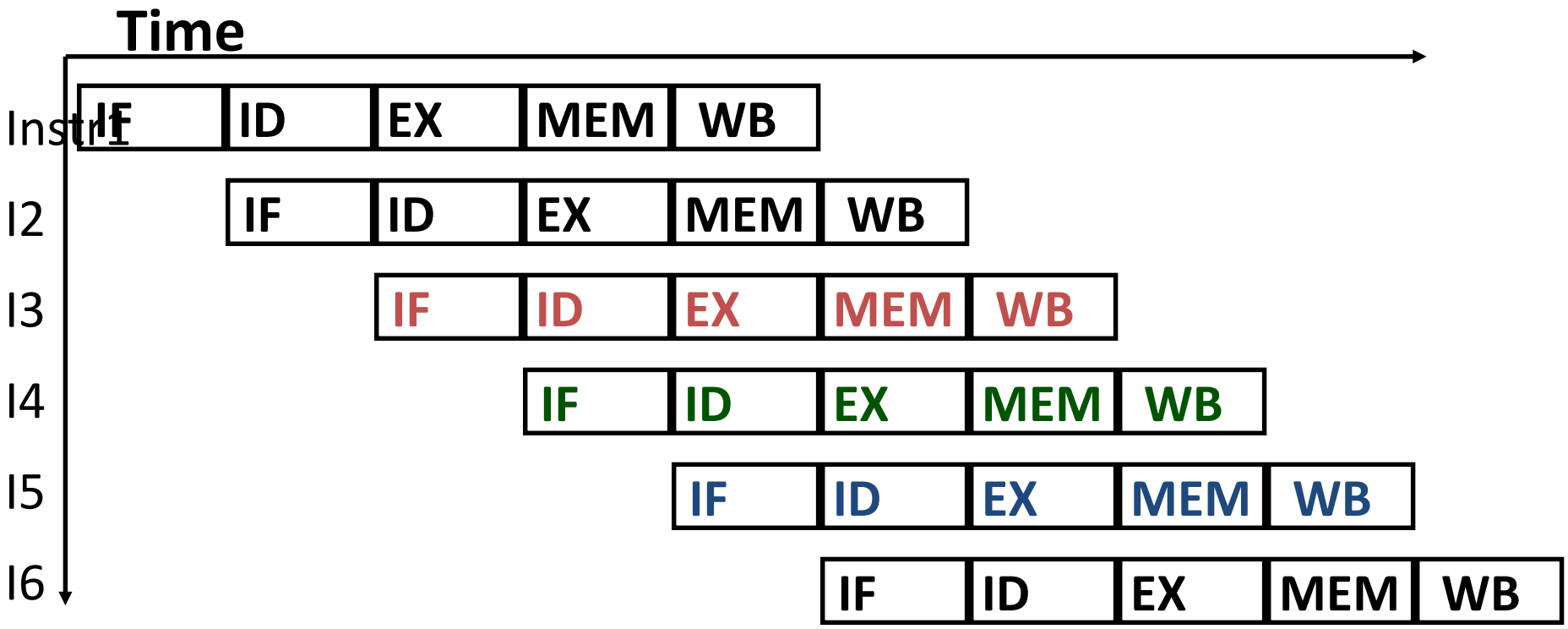


Corrected Datapath

- Now any instruction that writes to a register will work properly



Pipelined Execution Representation



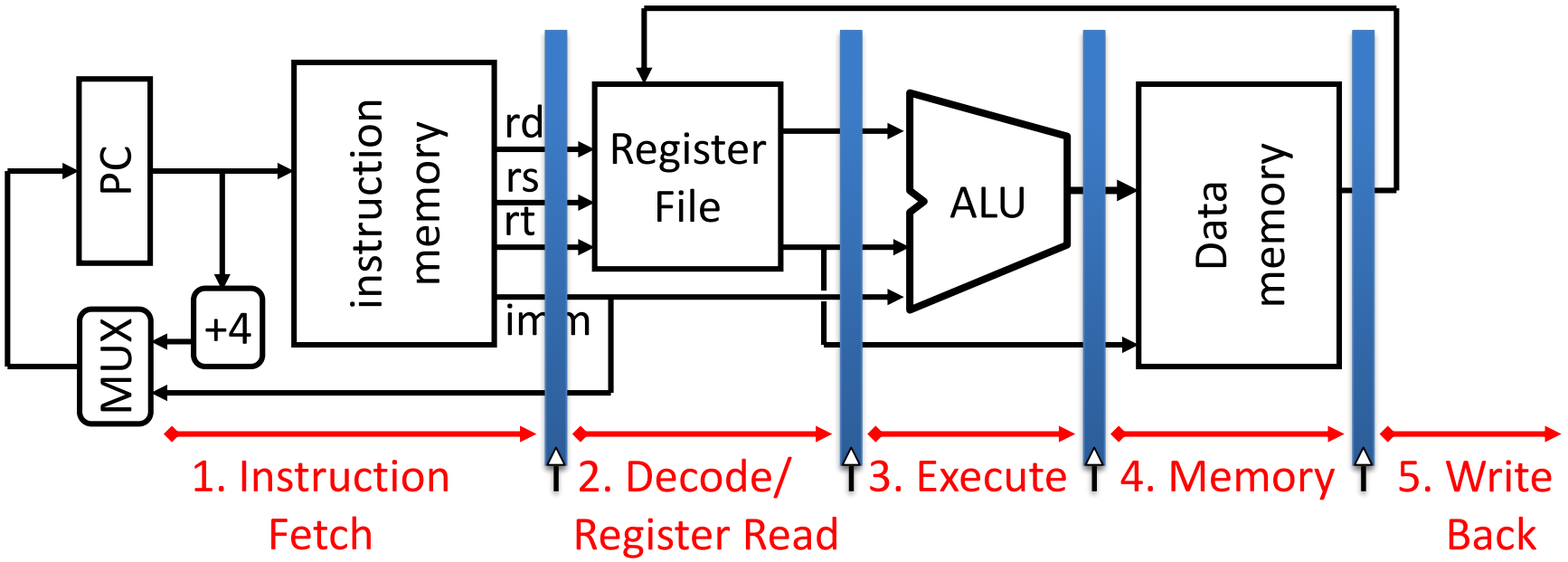
- Every instruction must take same number of steps, so some will idle
  - e.g. MEM stage for any arithmetic instruction

时钟驱动的流水线时空图

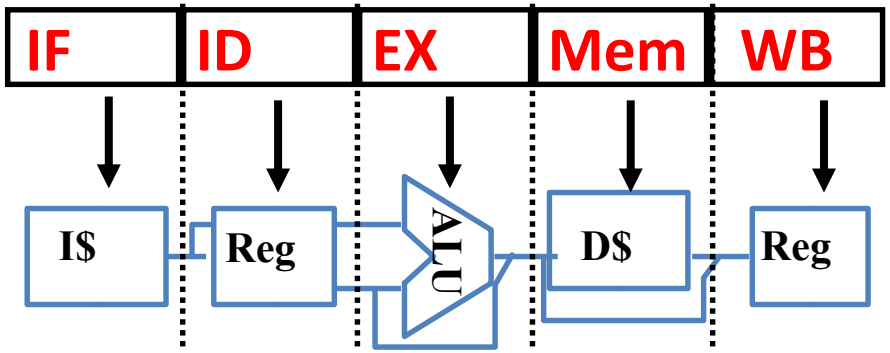
- 需精确分析指令/时间/流水线3者关系时
  - 指令何时处于何阶段
- 注意区分流水阶段与流水线寄存器的关系
  - EX级为例：ID/EX输出，驱动ALU，结果写入EX/MEM
- 可以看出，在clk5后，流水线全部充满
  - 所有部件都在执行指令：不同指令位于不同部件

相对PC的地址偏移	指令			IF级	ID级	EX级	MEM级	WB级	
		CLK	PC	IM	IF/ID	ID/EX	EX/MEM	MEM/WB	RF
0	Instr 1	↑ 1	0→4	Instr 1	Instr 1				
4	Instr 2	↑ 2	4→8	Instr 2	Instr 2	Instr 1			
8	Instr 3	↑ 3	12→12	Instr 3	Instr 3	Instr 2	Instr 1		
12	Instr 4	↑ 4	12→16	Instr 4	Instr 4	Instr 3	Instr 2	Instr 1	
16	Instr 5	↑ 5	16→20	Instr 5	Instr 5	Instr 4	Instr 3	Instr 2	Instr1
20	Instr 6	↑ 5	16→20	Instr6	Instr6	Instr5	Instr4	Instr3	Instr2

Graphical Pipeline Diagrams

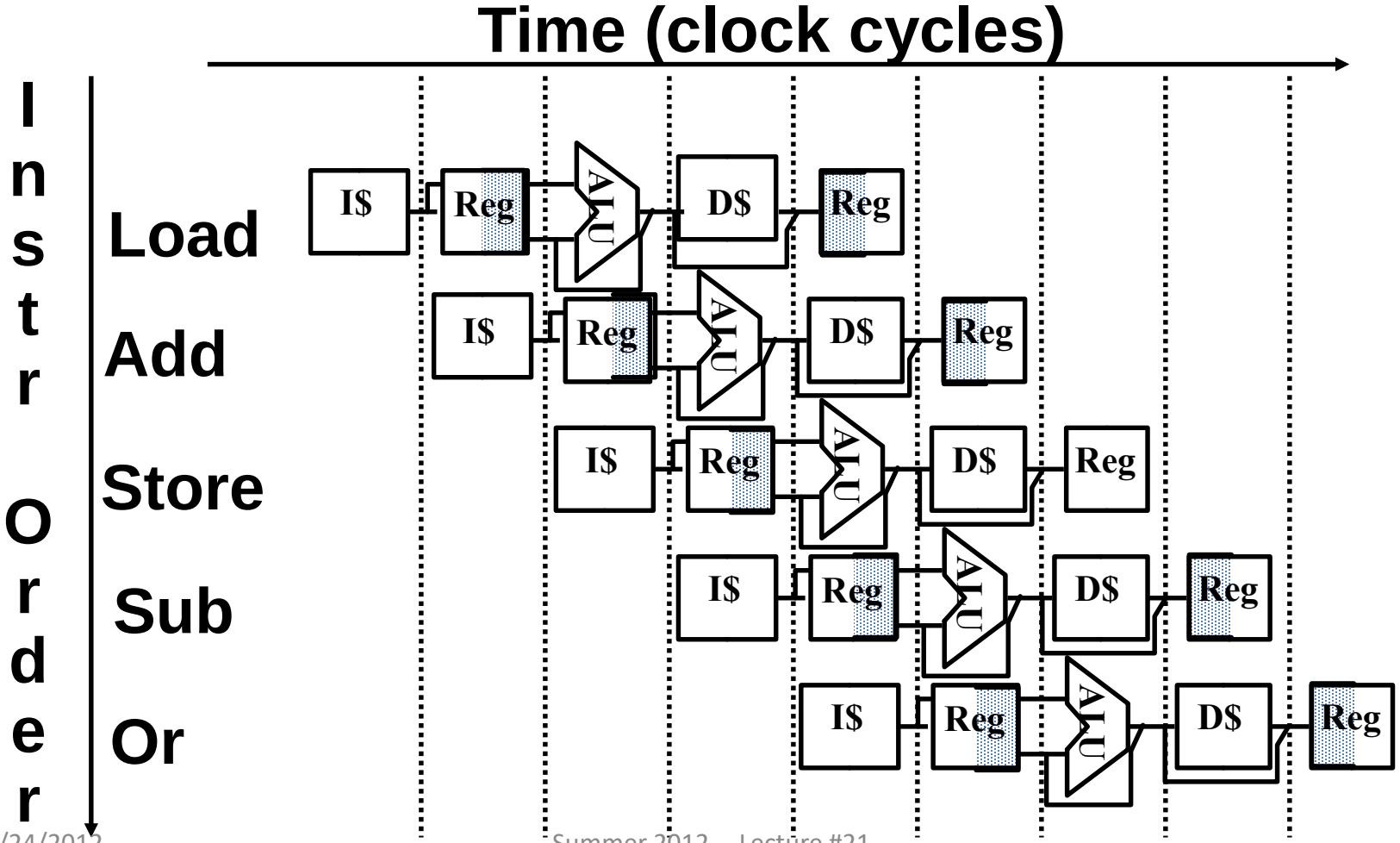


- Use datapath figure below to represent pipeline:



Graphical Pipeline Representation

- RegFile: right half is read, left half is write



Instruction Level Parallelism (ILP)

- Pipelining allows us to execute parts of multiple instructions at the same time using the same hardware!
  - This is known as *instruction level parallelism*

Pipeline Performance (1/2)

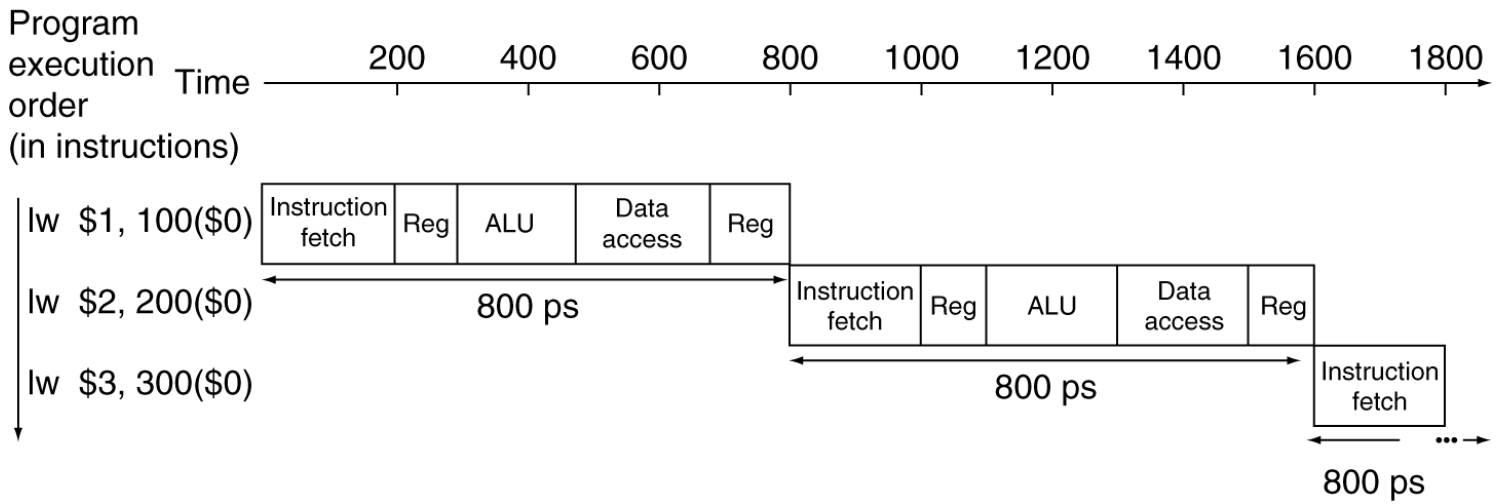
- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

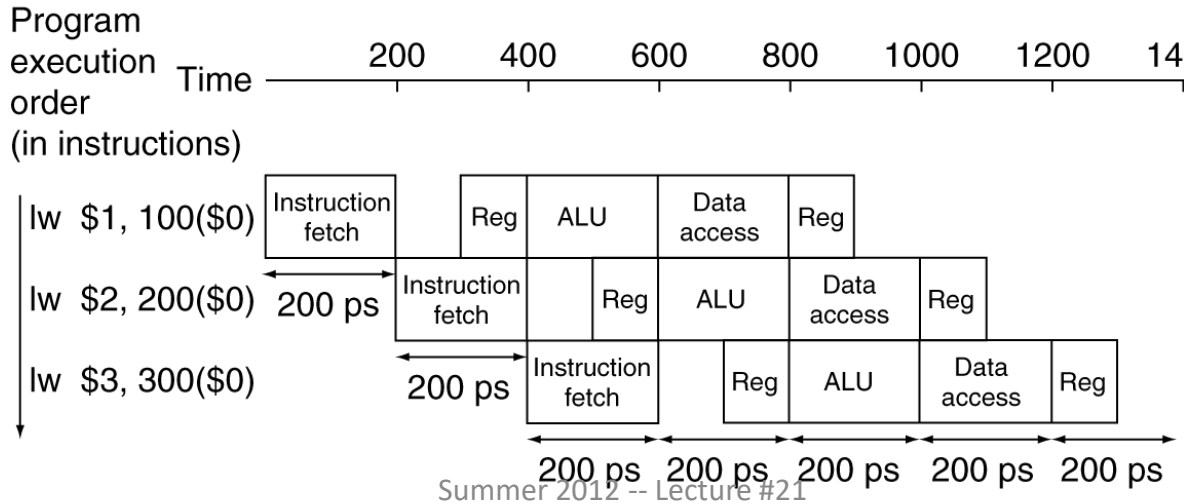
- What is pipelined clock rate?
  - Compare pipelined datapath with single-cycle datapath

Pipeline Performance (2/2)

Single-cycle
T_c = 800 ps



Pipelined
T_c = 200 ps



Pipeline Speedup

- Use T_c ("time between completion of instructions") to measure speedup
  - $T_{c,pipelined} \geq \frac{T_{c,single-cycle}}{\text{Number of stages}}$
  - Equality only achieved if stages are *balanced* (i.e. take the same amount of time)
- If not balanced, speedup is reduced
- Speedup due to increased throughput
  - Latency for each instruction does not decrease

Pipelining and ISA Design

- MIPS Instruction Set designed for pipelining!
- All instructions are 32-bits
  - Easier to fetch and decode in one cycle
- Few and regular instruction formats, 2 source register fields always in same place
  - Can decode and read registers in one step
- Memory operands only in Loads and Stores
  - Can calculate address 3rd stage, access memory 4th stage
- Alignment of memory operands
  - Memory access takes only one cycle

Pipelining Hazards

A *hazard* is a situation that prevents starting the next instruction in the next clock cycle

1) *Structural hazard*
   - A required resource is busy (e.g. needed in multiple stages)
2) *Data hazard*
   - Data dependency between instructions
   - Need to wait for previous instruction to complete its data read/write
3) *Control hazard*
   - Flow of execution depends on previous instruction

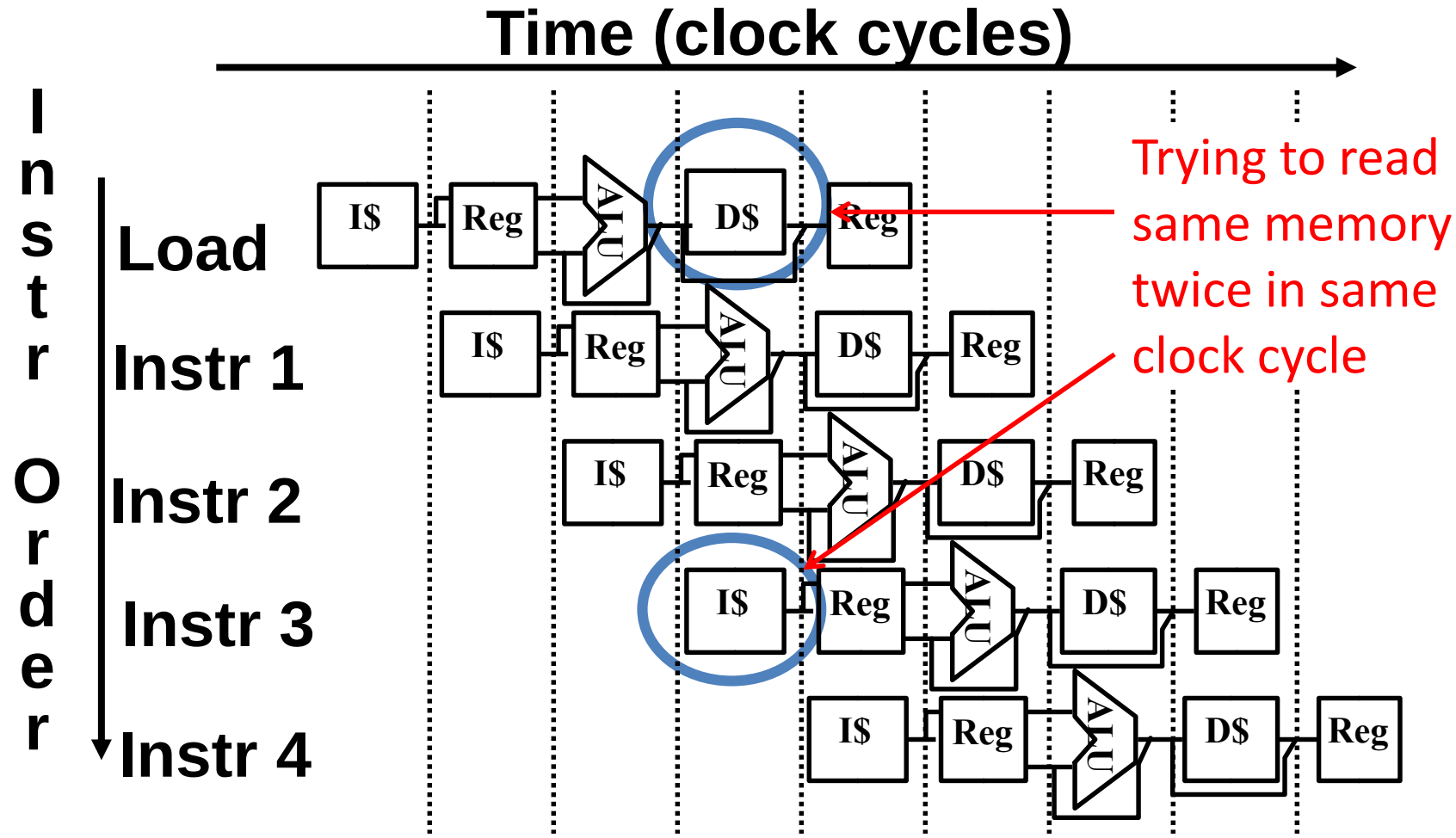
Agenda

- Structural Hazards
- Data Hazards
  - Forwarding
- Data Hazards (Continued)
  - Load Delay Slot
- Control Hazards
  - Branch and Jump Delay Slots
  - ~~Branch Prediction~~

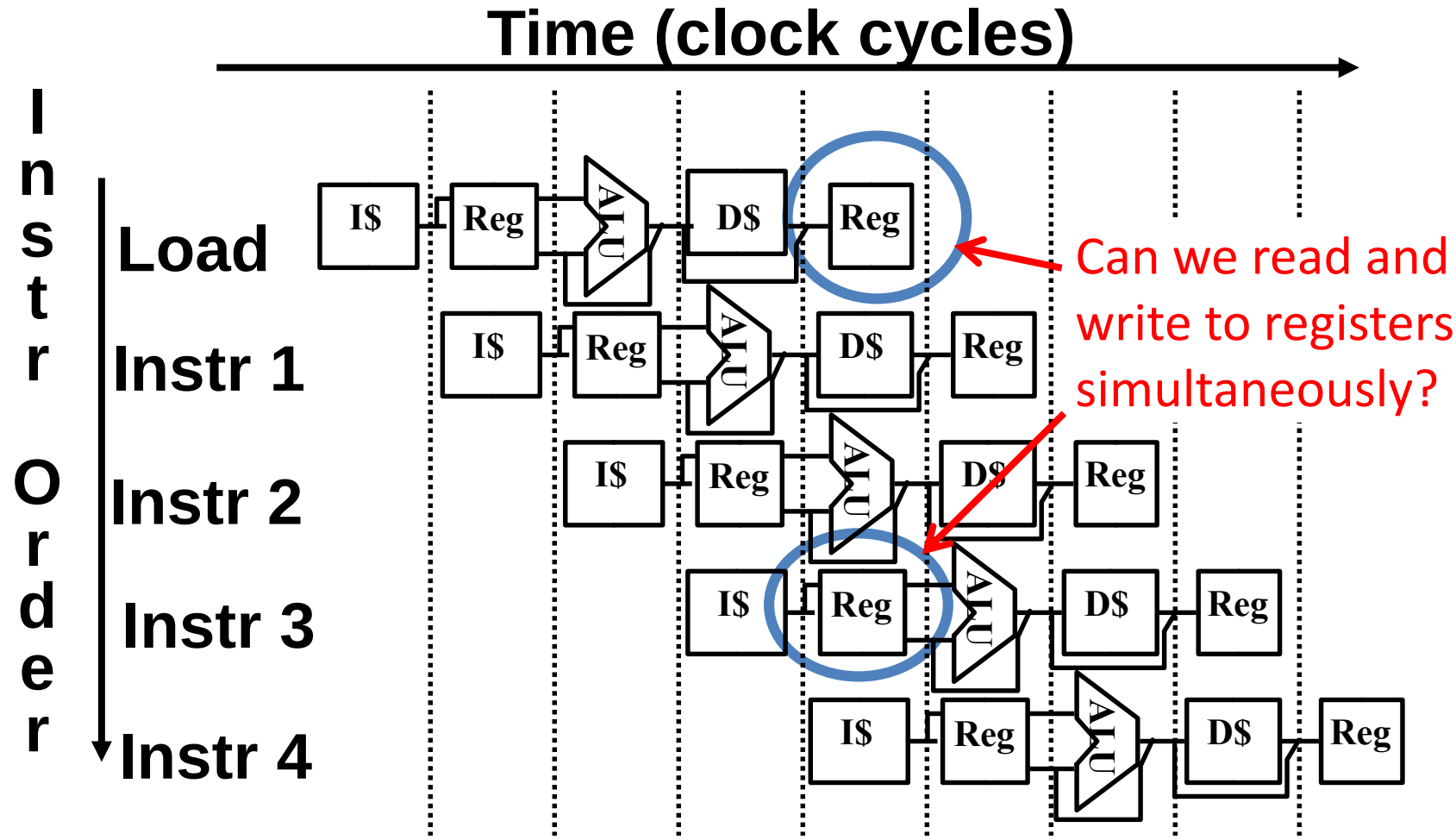
1. Structural Hazards

- Conflict for use of a resource
- MIPS pipeline with a single memory?
  - Load/Store requires memory access for data
  - Instruction fetch would have to *stall* for that cycle
    - Causes a pipeline "*bubble*"
- Hence, pipelined datapaths require separate instruction/data memories
  - Separate L1 I$ and L1 D$ take care of this

Structural Hazard #1: Single Memory



Structural Hazard #2: Registers (1/2)



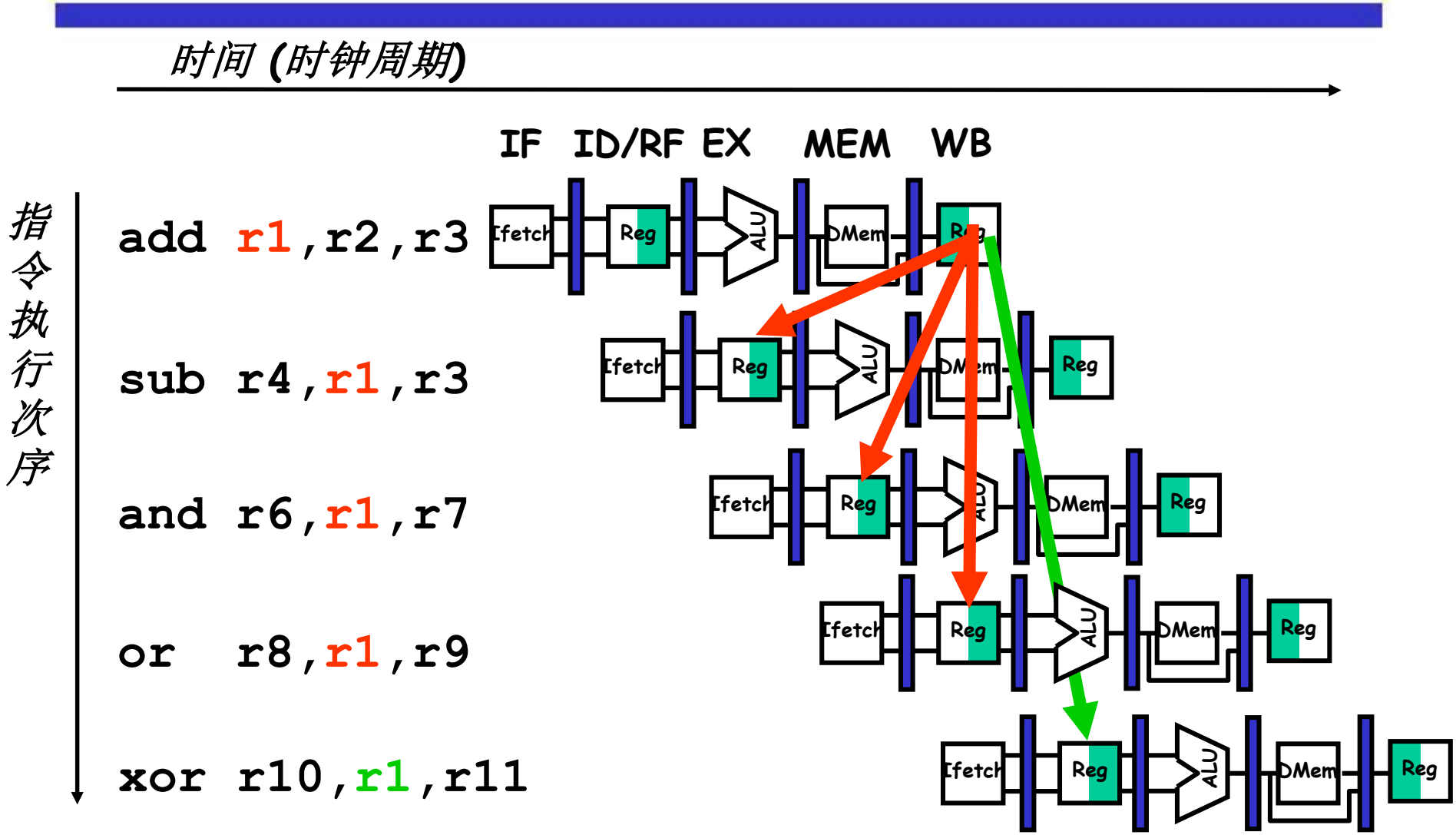
Structural Hazard #2: Registers (2/2)

- Two different solutions have been used:
  1) Split RegFile access in two: Write during 1st half and Read during 2nd half of each clock cycle
     - Possible because RegFile access is *VERY* fast (takes less than half the time of ALU stage)
  2) Build RegFile with **independent read and write ports**
- **Conclusion:** Read and Write to registers during same clock cycle is okay

Agenda

- Structural Hazards
- Data Hazards
  - Forwarding
- Data Hazards (Continued)
  - Load Delay Slot
- Control Hazards
  - Branch and Jump Delay Slots
  - Branch Prediction

数据冒险


时间 (时钟周期)
IF
ID/RF
EX
MEM
WB
指令执行次序
add r1,r2,r3
sub r4,r1,r3
and r6,r1,r7
or r8,r1,r9
xor r10,r1,r11
Ifetch
Reg
ALU
DMem
Reg

数据冒险

- 通过时钟驱动的流水线时空图，可以更好的标记具体执行状态
  - PC、IM以及各级流水线寄存器的变化

IF/ID	ID/EX	EX/MEM	MEM/WB

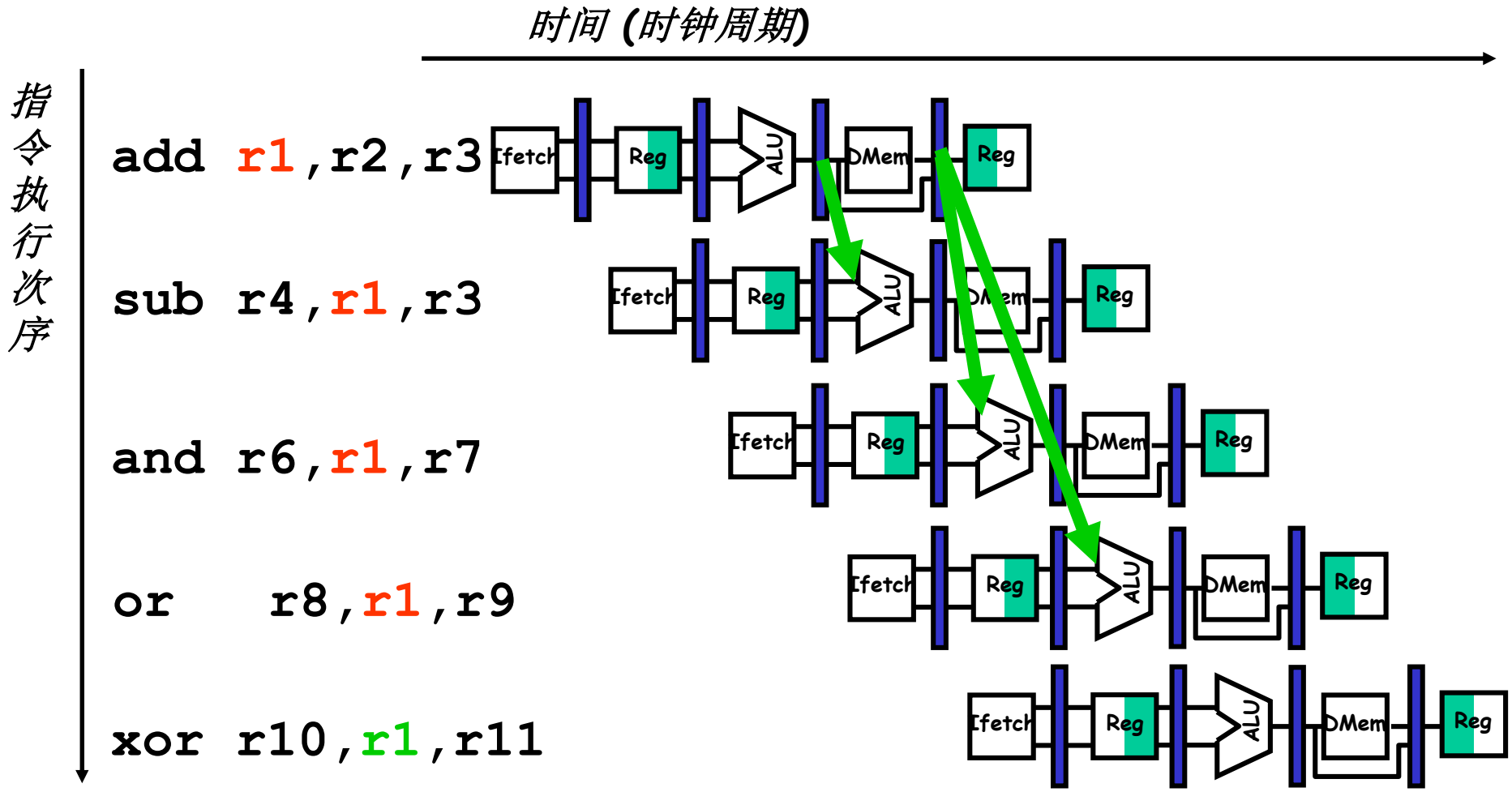
用更短的名字来命名流水线寄存器

D	E	M	W

地址	指令
0	add **r1**, r2, r3
4	sub r4, **r1**, r3
8	and r6, **r1**, r7
12	or r8, **r1**, r9
16	xor r10, **r1**, r11

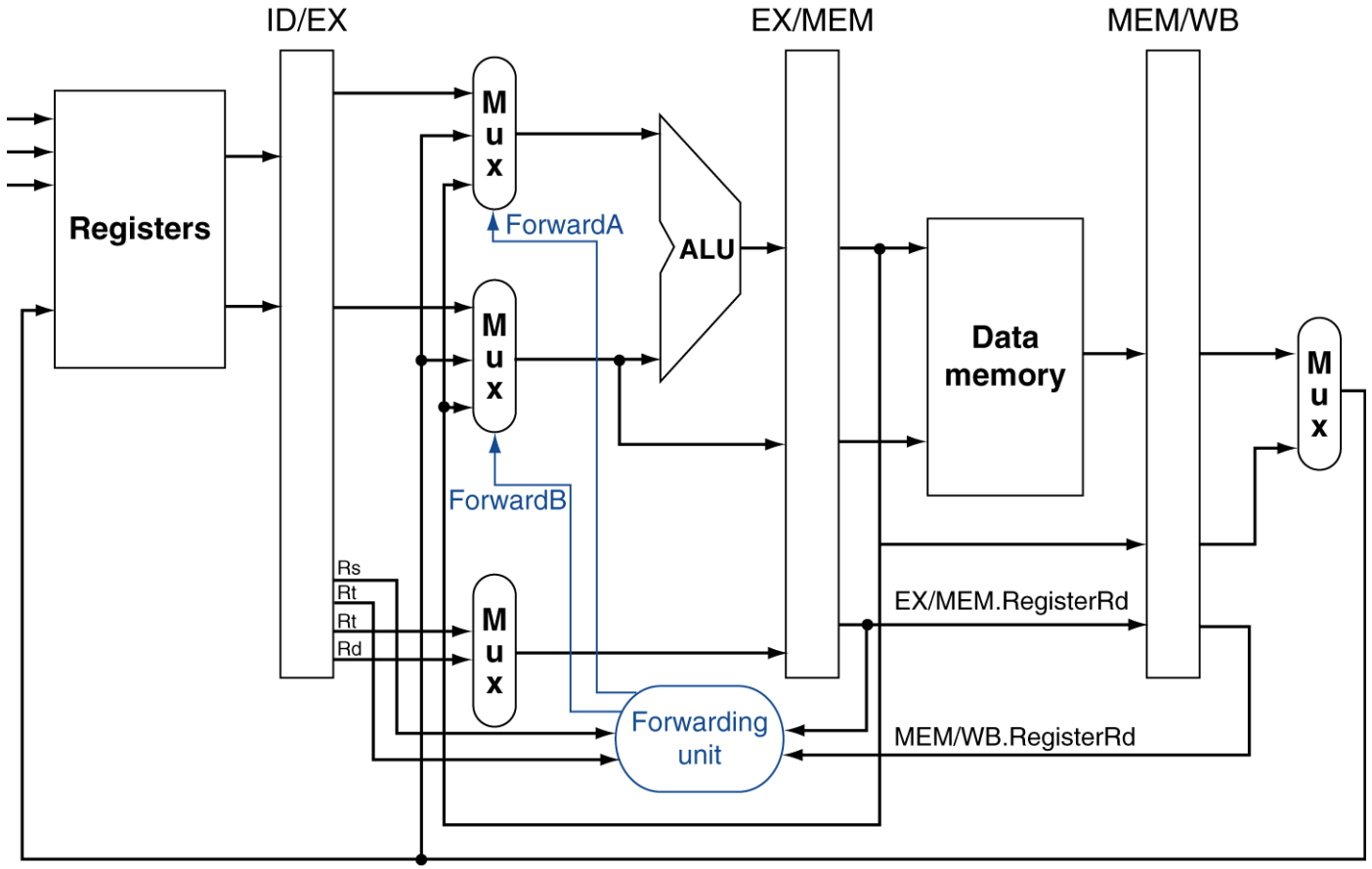
	PC+4		RF(读)	ALU	DM		
CLK	PC	IM	D	E	M	W	RF
↑1	0	add	add 写r1，无				旧值
	4	sub					
↑2	4	sub	sub 读r1，旧	add 写R1，无			旧值
	8	and					
↑3	8	and	and 读r1，旧	sub 读r1，旧	add 写R1，新		旧值
	12	or					
↑4	12	or	or 读r1，旧	and 读r1，旧	sub 读r1，旧	add 写R1，新	旧值
	16	xor					
↑5	16	xor	xor 读r1，新	or 读r1，旧	and 读r1，旧	sub 读r1，旧	新值
	20	xxx					

数据冒险解决策略—旁路


时间 (时钟周期)
指令执行次序
add r1,r2,r3
sub r4,r1,r3
and r6,r1,r7
or r8,r1,r9
xor r10,r1,r11
Ifetch
Reg
ALU
DMem
Reg

Datapath for Forwarding

- Handled by *forwarding unit*



Agenda

- Structural Hazards
- Data Hazards
  - Forwarding
- Data Hazards (Continued)
  - Load Delay Slot
- Control Hazards
  - Branch and Jump Delay Slots
  - Branch Prediction

上升沿-1：lw写入D级

- PC：指向sub指令的地址（PC ← PC + 4）
- IM：输出sub指令
- 3：还有3个cycle，从DM中才得到结果

地址	指令
0	lw $t0, 0($t1)
4	sub $t3, $t0, $t2
8	and $t5, $t0, $t4
12	or $t7, $t0, $t6
16	add $t1, $t2, $t3

	PC+4			RF(读)	ALU	DM		
CLK	PC	IM	D	E	M	W	RF
↴1	0	lw	lw				
	4	sub	写t0 3				

上升沿-2：sub写入D

- 分析： sub在1个cycle后就必须用$t0；但是lw还有2个cycle才能产生$t0的新值（此时尚未写入$t0）
- 结论：lw的$t0产生时间太晚，除了暂停sub，没有任何办法能解除这个冲突
  - 后续介绍如何暂停

地址	指令
0	lw $t0, 0($t1)
4	sub $t3, $t0, $t2
8	and $t5, $t0, $t4
12	or $t7, $t0, $t6
16	add $t1, $t2, $t3

	PC+4			RF(读)	ALU	DM		
CLK	PC	IM	D	E	M	W	RF
↴1	0	lw	lw				
	4	sub	写t0 3				
↴2	4	sub	sub	lw			
	8	and	读t0 1	写t0 2			

上升沿-3：lw进入M

- sub还有1个cycle使用t0，lw还有1个cycle产生新t0
- 冲突分析：冲突解除
  - 转发机制将在clk4时可以发挥作用

```
地址        指令
 0  lw   $t0, 0($t1)
 4  sub  $t3, $t0, $t2
 8  and  $t5, $t0, $t4
12  or   $t7, $t0, $t6
16  add  $t1, $t2, $t3
```

	PC+4			RF(读)	ALU	DM		
CLK	PC	IM	D	E	M	W	RF
↴1	0	lw	lw				
	4	sub	写t0 3				
↴2	4	sub	sub	lw			
	8	and	读t0 1	写t0 2			
↴3	8	and	sub	nop	lw		
	8	and	读t0 1		写t0 1		

上升沿-4（1/2）：sub到达E

- lw：从DM中读出数据并写入W。t0新值产生了。
- 执行：控制ALU的转发MUX，使之选择来自W级的新t0
  - ALU用新t0完成正确的计算

```
地址        指令
 0  lw   $t0, 0($t1)
 4  sub  $t3, $t0, $t2
 8  and  $t5, $t0, $t4
12  or   $t7, $t0, $t6
16  add  $t1, $t2, $t3
```

		PC+4			RF(读)	ALU	DM		
CLK	PC	IM	D		E	新t0	M	W	RF
↑1	0	lw	lw						
	4	sub	写t0	3					
↑2	4	sub	sub		lw				
	8	and	读t0	1	写t0	2			
↑3	8	and	sub		nop		lw		
	8	and	读t0	1			写t0	1	
↑4					sub		nop	lw	
					读t0	0		新t0	0

上升沿-4（2/2）：and写入D

- 注意：{and@D，lw@W}同样需要转发
- Q：如果没有W至D的转发，则and指令将无法正确执行。为什么？

```
地址        指令
 0  lw   $t0, 0($t1)
 4  sub  $t3, $t0, $t2
 8  and  $t5, $t0, $t4
12  or   $t7, $t0, $t6
16  add  $t1, $t2, $t3
```

		PC+4		RF(读)	ALU	DM		
CLK	PC	IM	D	E	M	W	RF
↑1	0	lw	lw		新t0		
	4	sub	写t0 3				
↑2	4	sub	sub	lw			
	8	and	读t0 1	写t0 2			
↑3	8	and	sub	nop	lw		
	8	and	读t0 1		写t0 1		
↑4	8	and	and	sub	nop	lw	
	12	or	读t0 1	读t0 0		新t0 0	

上升沿-4：and写入D

- {sub，lw}：控制MUX，使得W级的新t0转发至ALU
- {and，lw}：同样需要转发
  - lw：尚未将新t0写入RF

Q：
没有W至D的转发，则and指令将无法正确执行。为什么？

```
地址        指令
 0  lw   $t0, 0($t1)
 4  sub  $t3, $t0, $t2
 8  and  $t5, $t0, $t4
12  or   $t7, $t0, $t6
16  add  $t1, $t2, $t3
```

	PC+4			RF(读)	ALU	DM	
CLK	PC	IM	D	E	M	W	RF
↑1	0	lw	lw				
	4	sub	写t0 3				
↑2	4	sub	sub	lw			
	8	and	读t0 1	写t0 2			
↑3	8	and	sub	nop	lw		
	8	and	读t0 1		写t0 1		
↑4	8	and	and	sub	nop	lw	
	12	or	读t0 1	新t0 0		新t0 0	

上升沿-5：lw写回

- lw：结果写回至RF
- or：写入D；然后开始读RF
  - 由于lw已经完成了回写，因此or读出的t0是最新的值

```
地址        指令
 0  lw   $t0, 0($t1)
 4  sub  $t3, $t0, $t2
 8  and  $t5, $t0, $t4
12  or   $t7, $t0, $t6
16  add  $t1, $t2, $t3
```

	PC+4			RF(读)	ALU	DM		
CLK	PC	IM	D	E	M	W	RF
↑1	0	lw	lw				
	4	sub	写t0 3				
↑2	4	sub	sub	lw			
	8	and	读t0 1	写t0 2			
↑3	8	and	sub	nop	lw		
	8	and	读t0 1		写t0 1		
↑4	8	and	and	sub	nop	lw	
	12	or	读t0 1	用t0 0		写t0 0	
↑5	12	or	or	and	sub	nop	新t0
	16	add	读t0 1	用t0 0			

Data Hazard: Loads (3/4)

- Stall is equivalent to `nop`

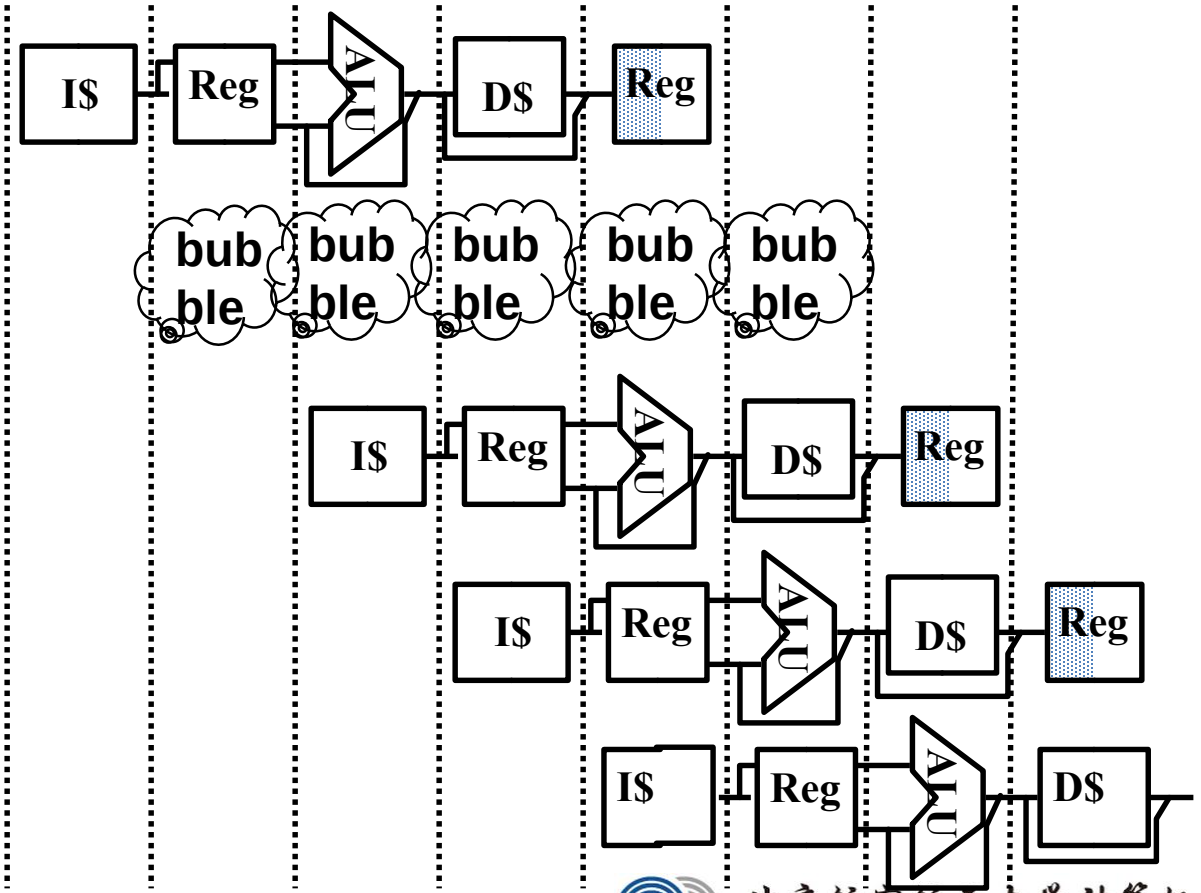
lw $t0, 0($t1)

nop

sub $t3,$t0,$t2

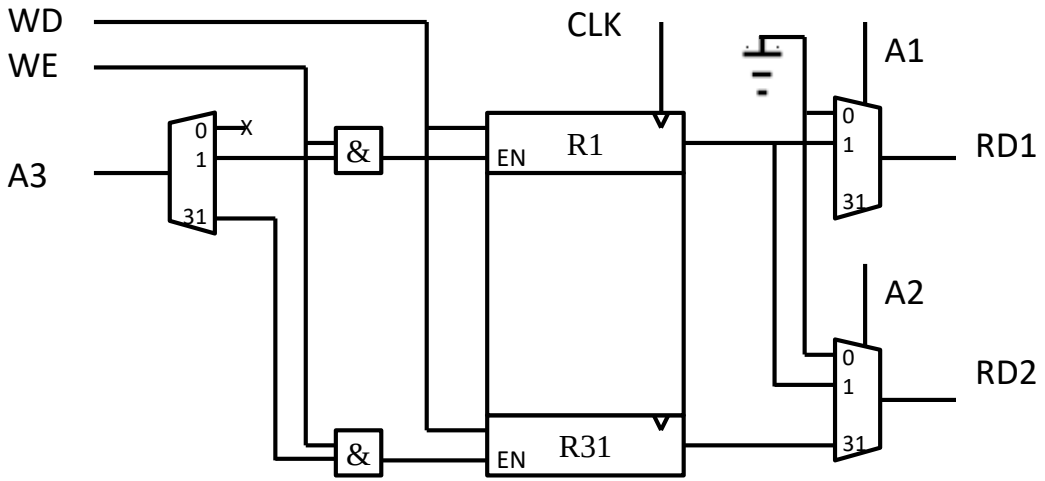
and $t5,$t0,$t4

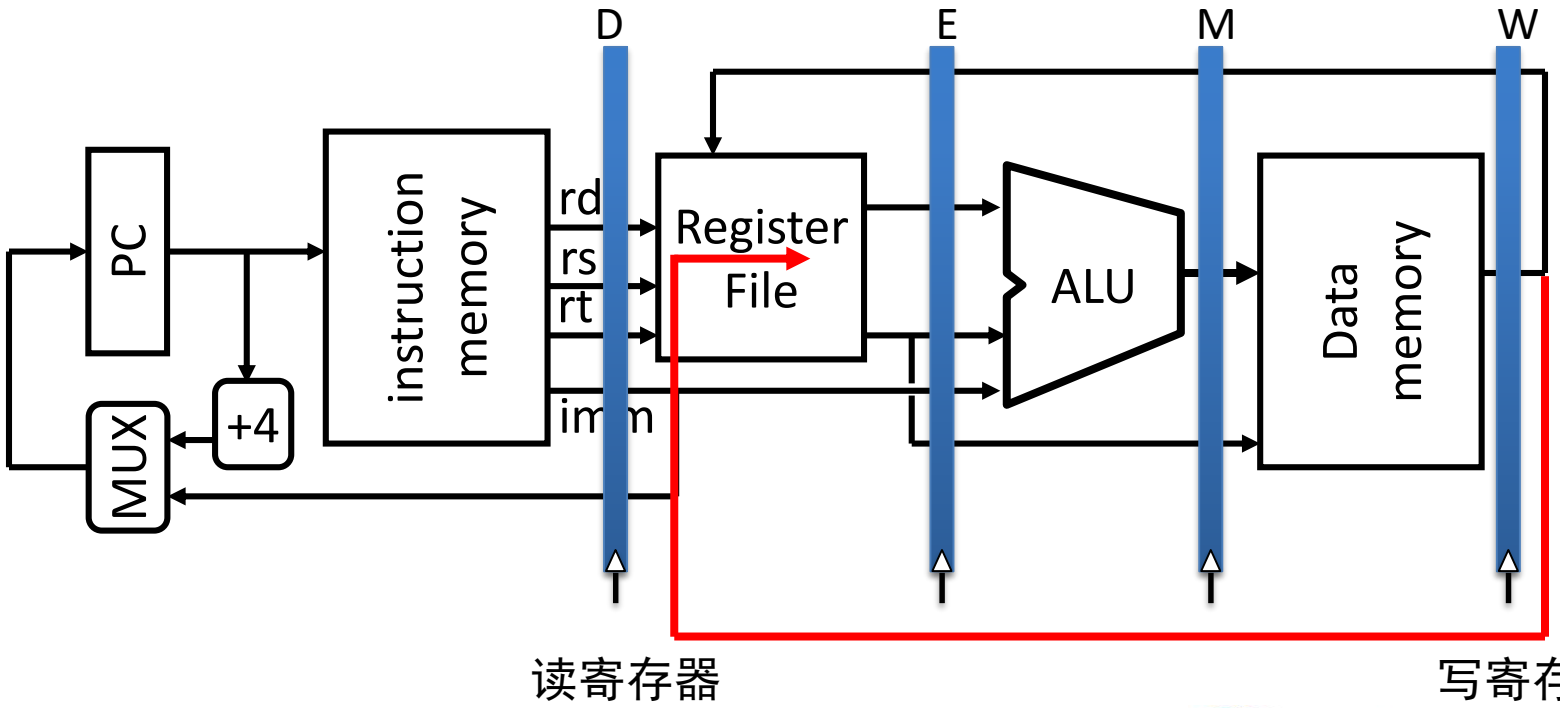
or $t7,$t0,$t6



RF的内部转发设计

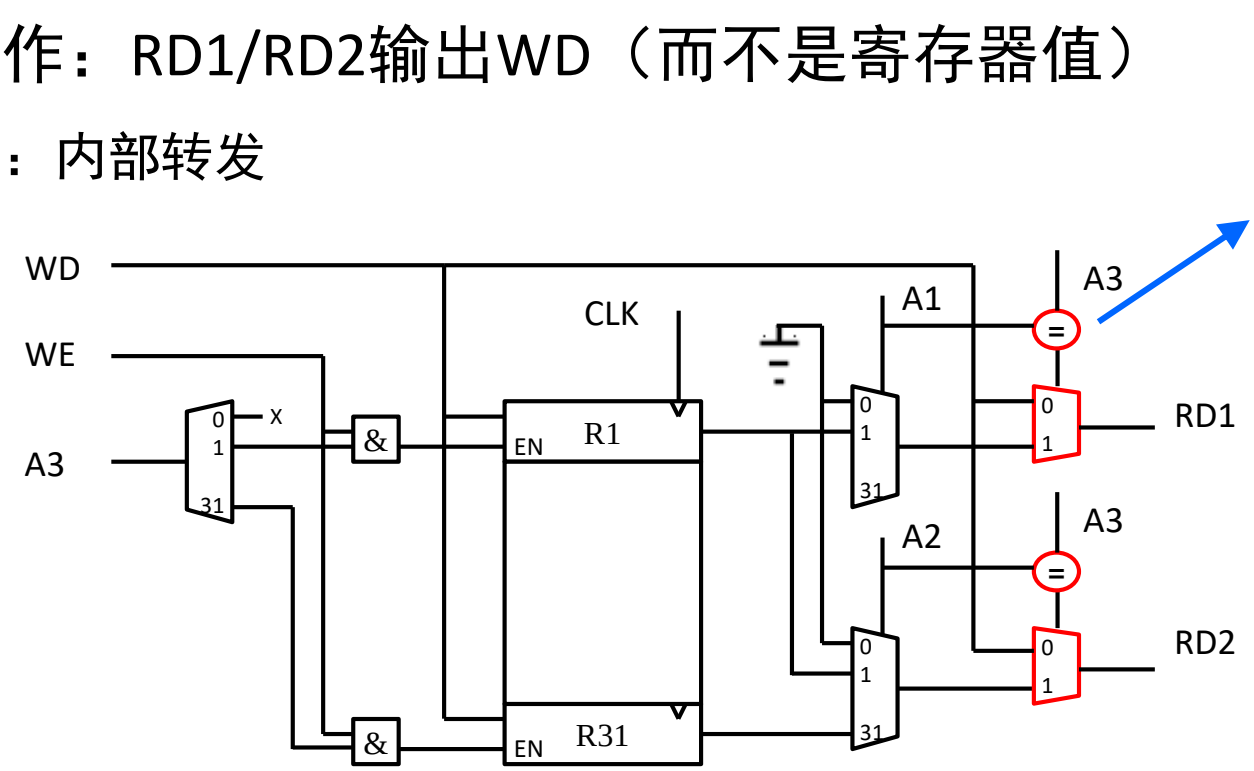
- 场景：当W级写RF，而D读RF时
- 如果没有转发，则虽然RF被更新了，但旧值进入了E





RF的内部转发设计

- 判断条件：写入寄存器与读出寄存器相同
  - A3 == A1 或者 A3 == A2
- 执行操作：RD1/RD2输出WD（而不是寄存器值）
  - 本质：内部转发



- Q：是否有遗漏条件？
- H：当前虽然不是写入操作，但是A3恰好等A1/A2！

把冲突指令发射直至需要的时候暂停？

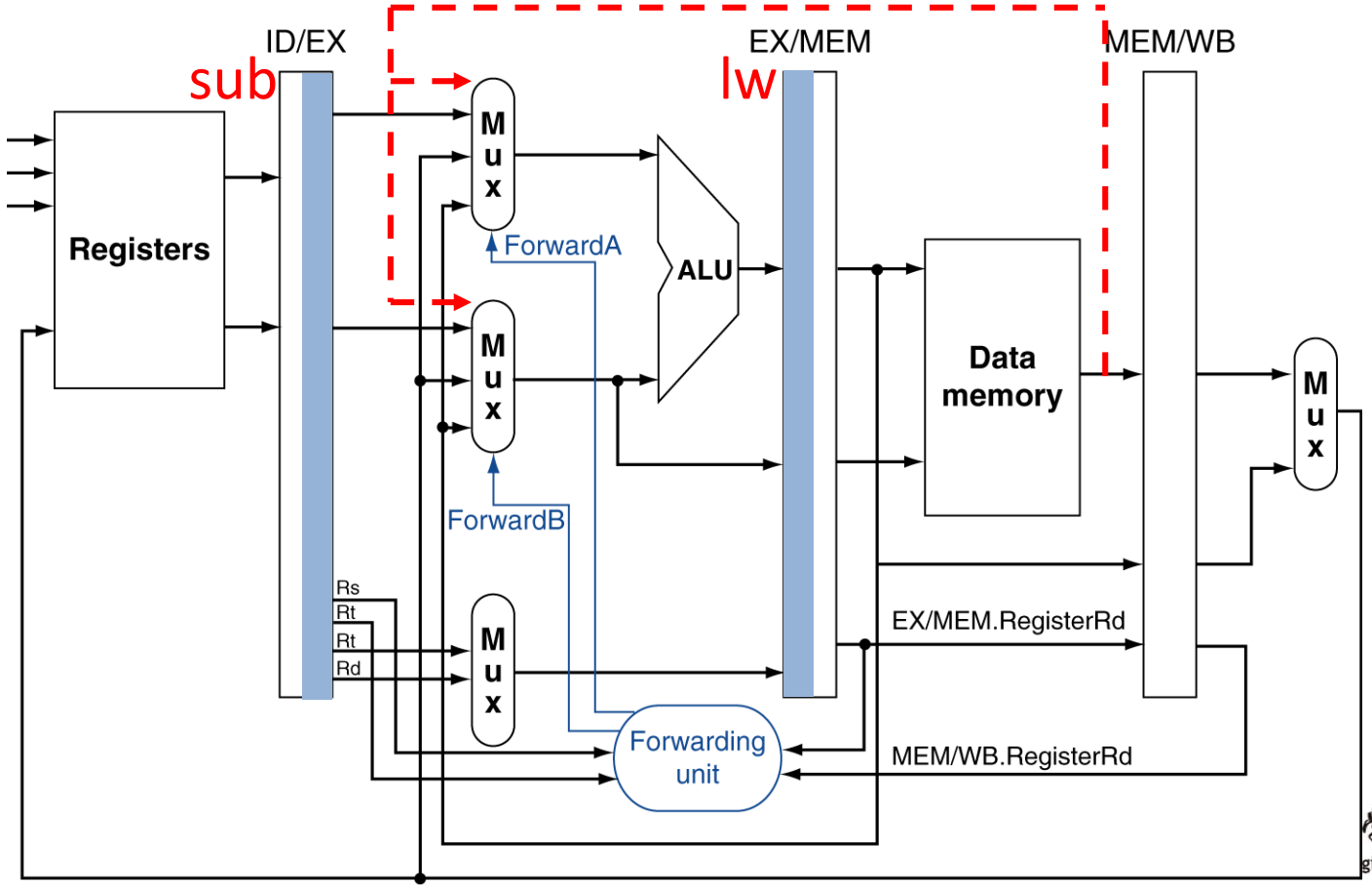
- 思路：让冲突指令继续执行，直至必须暂停
- 分析：暂停的周期是一样多的
- 结论：对于单发射顺序执行流水线，该方案没有任何性能改进；反而增加冲突分析的复杂度
  - 在IF/ID阶段进行冲突分析更好
  - 进一步的结论：但该方案在多发射/乱序执行架构中有意义

地址	指令
0	lw $t0, 0($t1)
4	sub $t3, $t0, $t2
8	and $t5, $t0, $t4
12	or $t7, $t0, $t6
16	add $t1, $t2, $t3

	PC+4			RF(读)	ALU	DM		
CLK	PC	IM	D	E	M	W	RF
↑1	0	lw	lw				
	4	sub	写t0 3				
↑2	4	sub	sub	lw			
	8	and	读t0 1	写t0 2			
↑3	8	and	and	sub	lw		
	8	and	读t0 1	读t0 0	写t0 1		
↑4	8	and	and	sub	nop	lw	
	12	or	读t0 1	用t0 0		写t0 0	
↑5	12	or	or	and	sub	nop	新t0
	16	add	读t0 1	用t0 0			

不合理的转发设计

- Q：如果设置从DM到ALU输入的转发，这个设计优劣如何？
  - 设计初衷：将DM读出数据提前1个clock转发至ALU，从而消除lw指令导致的数据相关，无需插入NOP

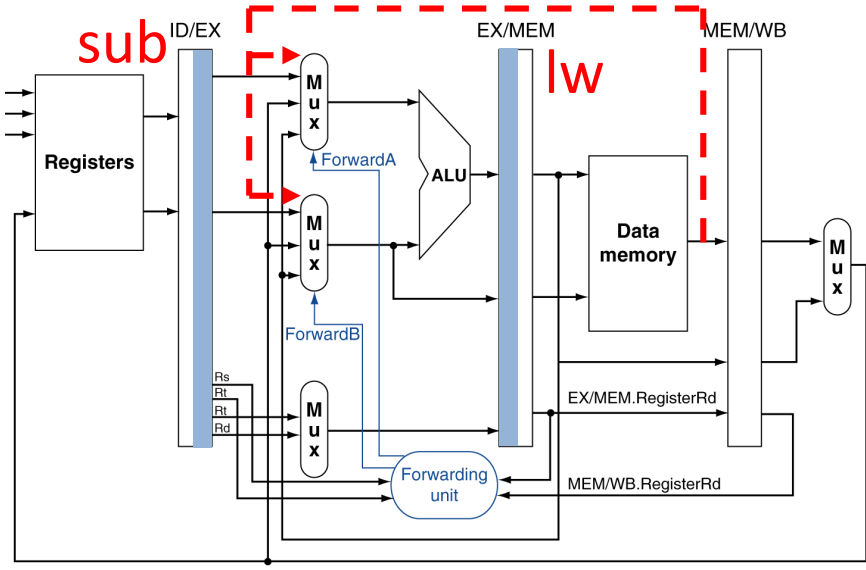


不合理的转发设计

- A：功能虽然正确，但CPU时钟频率大幅度降低
  - 原设计： f = 5GHz
    - 各阶段最大延迟为200ps
  - 新设计： f = 2.5GHz
    - EX阶段$_{修改后}$ = ALU延迟 + DM延迟 = 400ps
    - EX阶段延迟成为最大延迟

警惕：木桶原理！

流水线各阶段延迟不均衡，将导致流水线性能严重下降



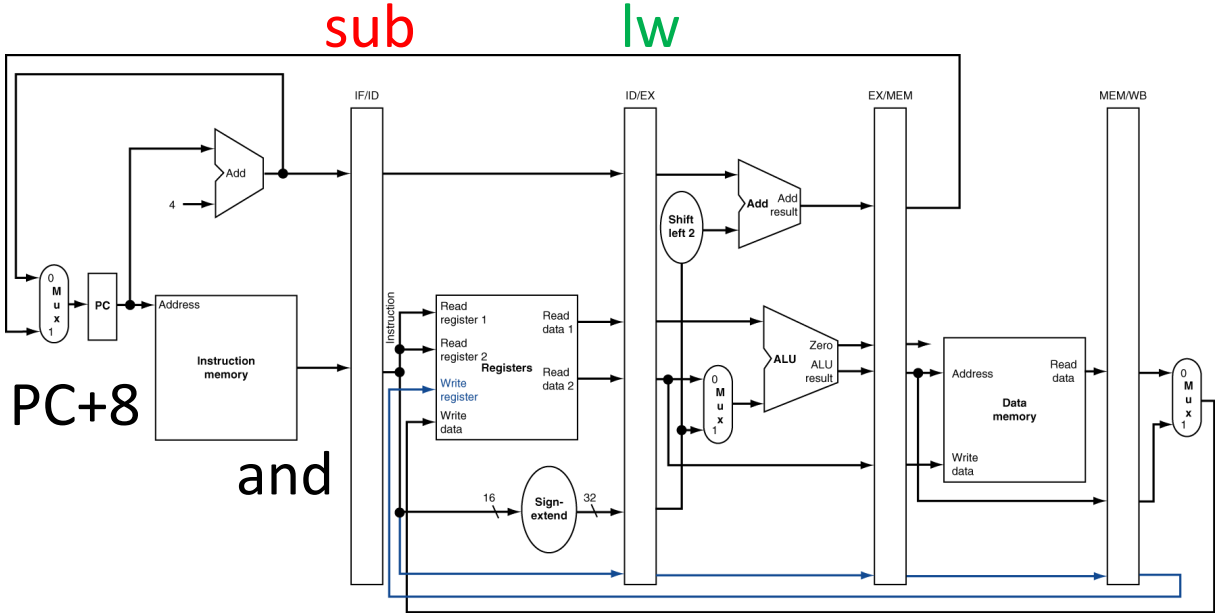
前面PPT的数据

Instr fetch	Register read	ALU op	Memory access	Register write
200ps	100 ps	200ps	200ps	100 ps

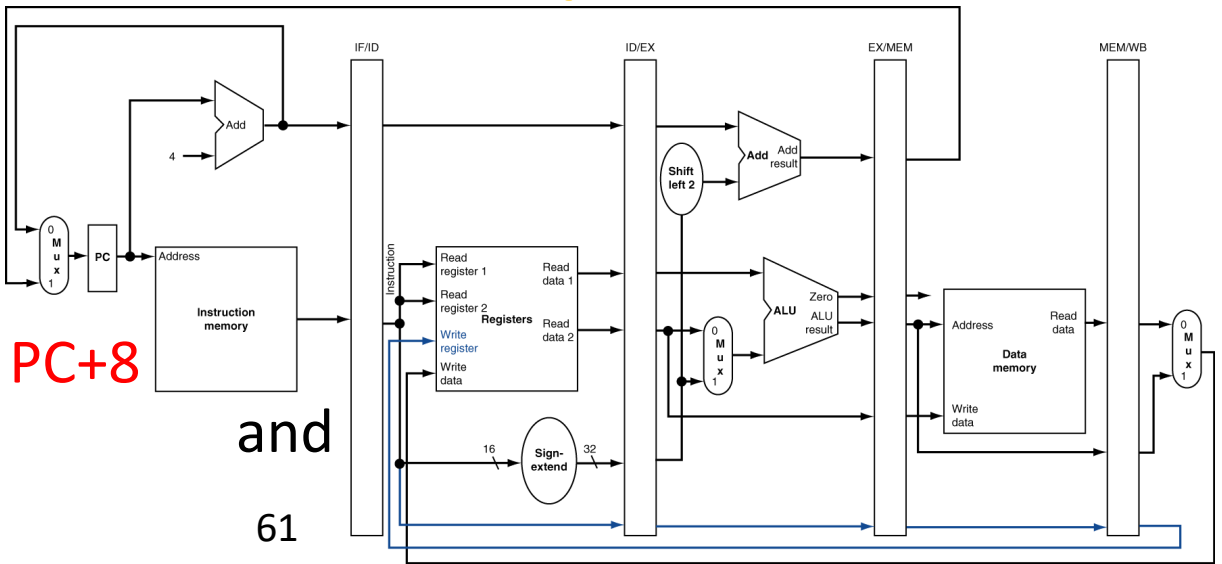
插入NOP指令

Cycle N

地址	指令
0	`lw $t0, 0($t1)`
4	`sub $t3, $t0, $t2`
8	`and $t5, $t0, $t4`
12	`or $t7, $t0, $t6`
16	`add $t1, $t2, $t3`



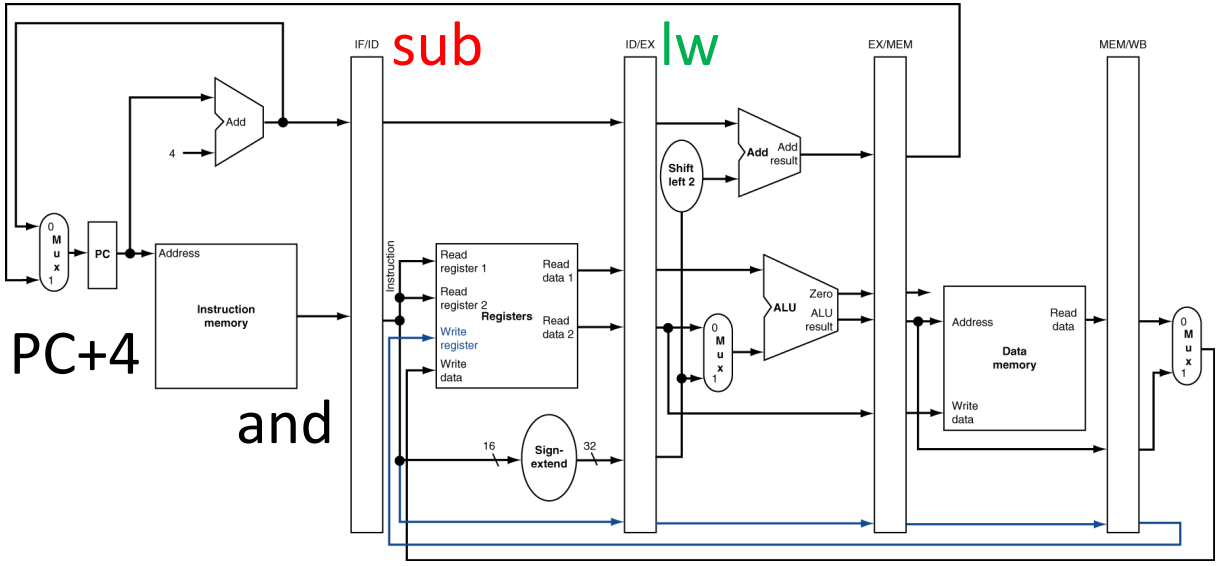
Cycle N+1



插入NOP指令

- 检测条件：IF/ID的前序是lw指令，并且lw的rt寄存器与IF/ID的rs或rt相同
- 执行动作：
  - ①冻结IF/ID：sub继续被保存
  - ②清除ID/EX：指令全为0，等价于插入NOP
  - ③禁止PC：防止PC继续计数，PC应保持为PC+4

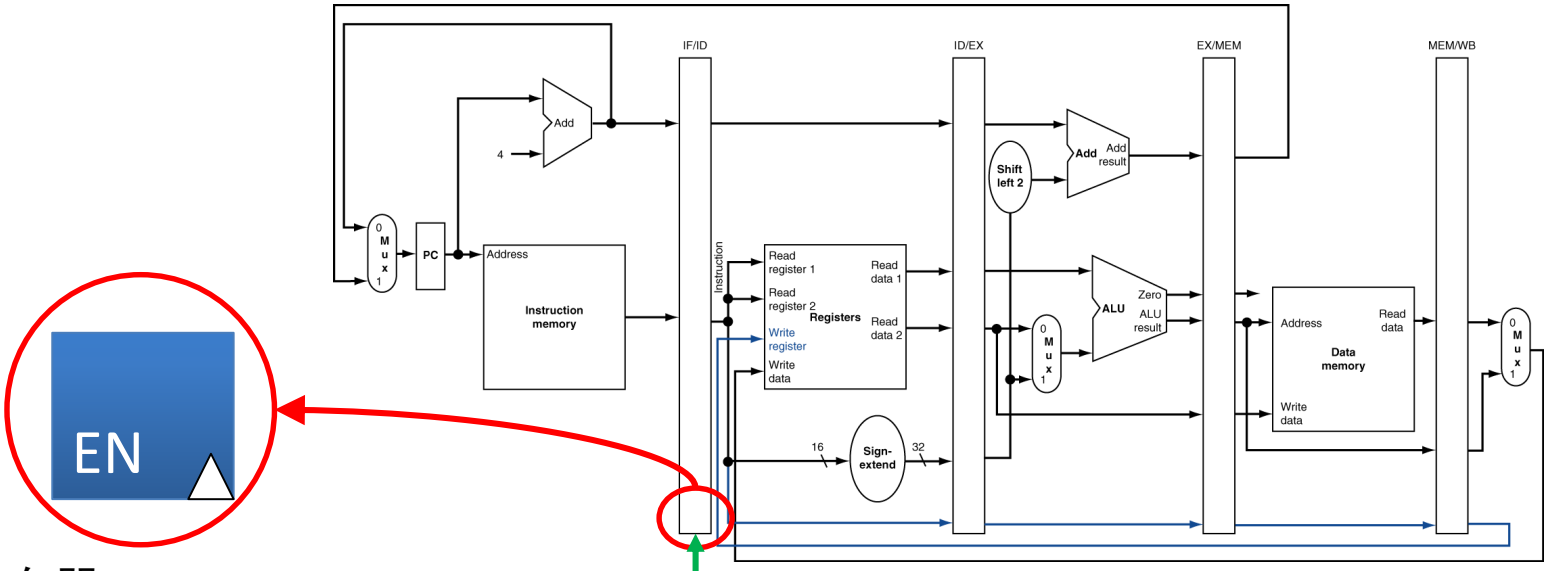
地址	指令
0	`lw $t0, 0($t1)`
4	`sub $t3, $t0, $t2`
8	`and $t5, $t0, $t4`
12	`or $t7, $t0, $t6`
16	`add $t1, $t2, $t3`



插入NOP指令：IF/ID增加使能

使能型寄存器

- 执行动作：
  - ①冻结IF/ID：sub继续被保存
  - ②清除ID/EX：指令全为0，等价于插入NOP
  - ③禁止PC：防止PC继续计数，PC应保持为PC+4
- 数据通路：将IF/ID修改为使能型寄存器
- 控制系统：增加IF/ID.en控制信号
  - 当IF/ID.en为0时，IF/ID在下个clock上升沿到来时保持不变

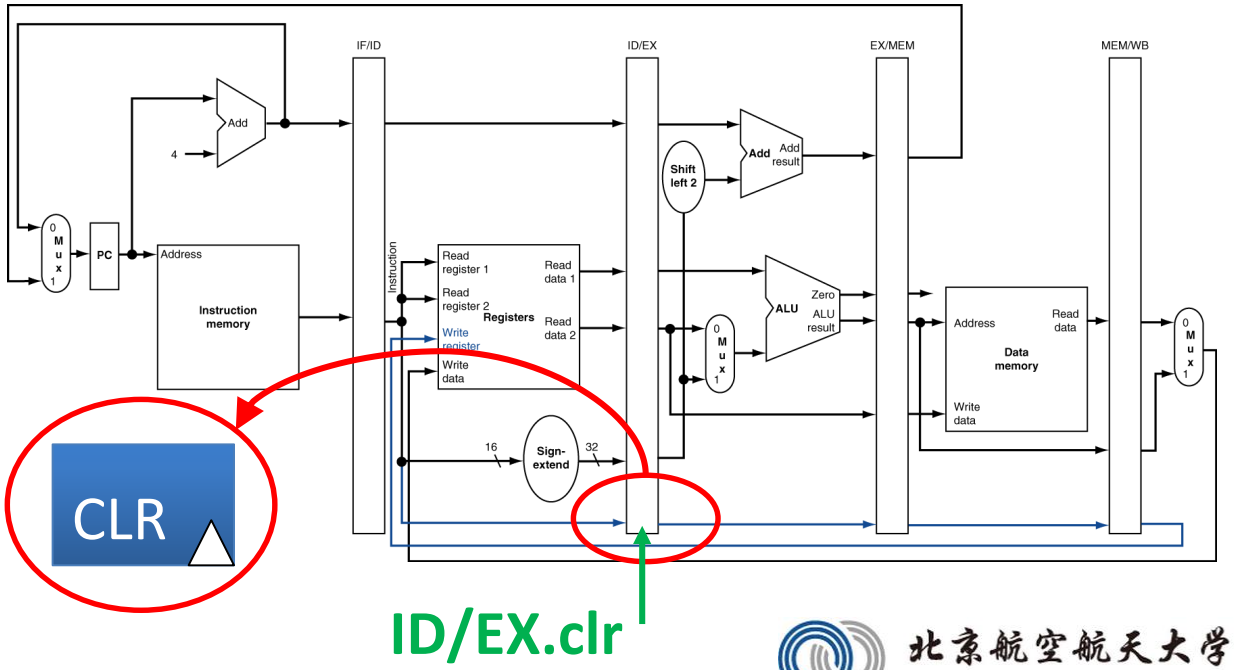


使能型寄存器

插入NOP指令：ID/EX增加清除

复位型寄存器

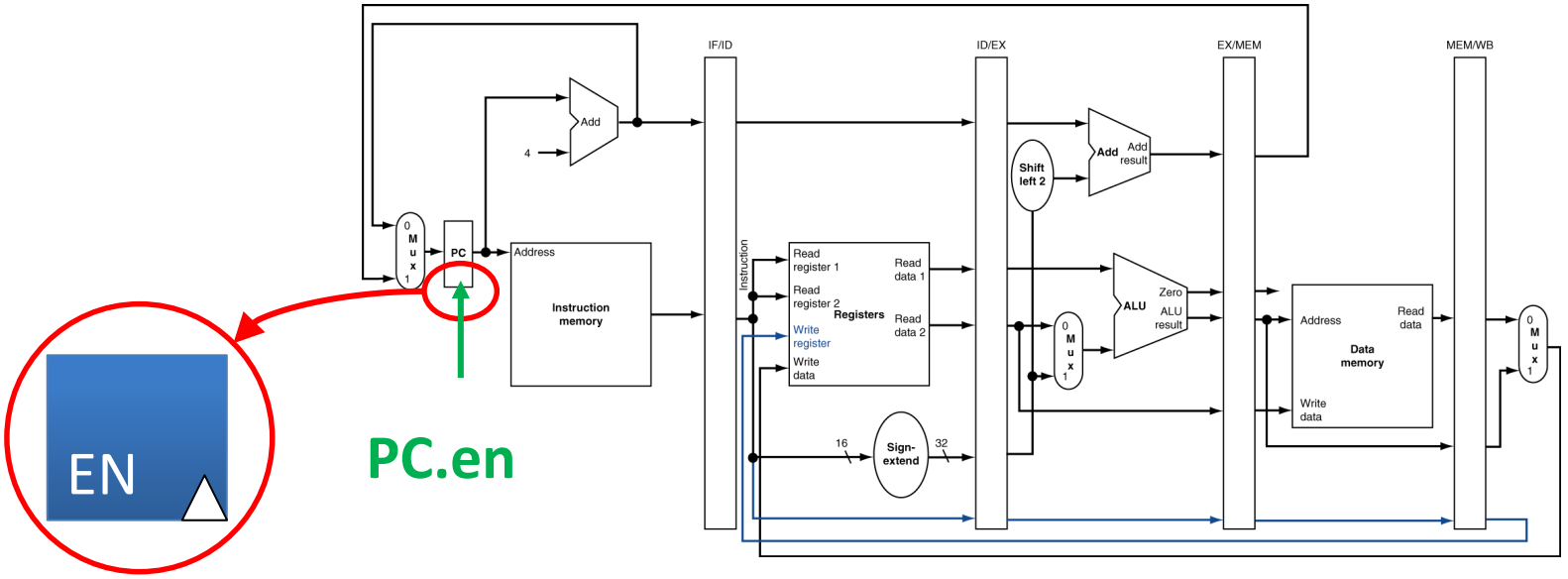
- 执行动作：
  - ①冻结IF/ID：sub继续被保存
  - ②清除ID/EX：指令全为0，等价于插入NOP
  - ③禁止PC：防止PC继续计数，PC应保持为PC+4
- 数据通路：将ID/EX修改为复位型寄存器
- 控制系统：增加ID/EX.clr控制信号
  - 当ID/EX.clr为0时，ID/EX在下个clock上升沿到来时被清除为0



插入NOP指令：PC增加使能

使能型寄存器

- 执行动作：
  - ①冻结IF/ID：sub继续被保存
  - ②清除ID/EX：指令全为0，等价于插入NOP
  - ③禁止PC：防止PC继续计数，PC应保持为PC+4
- 数据通路：将PC修改为使能型寄存器
- 控制系统：增加PC.en控制信号
  - 当PC.en为0时，PC在下个clock上升沿到来时保持不变



如何插入NOP指令？

- lw冒险处理示例伪代码
- 注意：时序关系
  - 各信号在clk2上升沿后有效
  - NOP是在clk3上升沿后发生，即寄存器值在clk3上升沿到来时发生变化(或保持不变)

```
if (ID/EX.MemRead) &
   ((ID/EX.rt == IF/ID.rs) |
    (ID/EX.rt == IF/ID.rt)
    IF/ID.en ← 禁止
    ID/EX.clr ← 清除
    PC.en ← 禁止
```

地址	指令
0	lw $t0, 0($t1)
4	sub $t3, $t0, $t2
8	and $t5, $t0, $t4
12	or $t7, $t0, $t6
16	add $t1, $t2, $t3

			RF(读)	ALU	DM		
CLK	PC	IM	D	E	M	W	RF
↑ 1	0→4	lw→sub	**lw**				
↑ 2	4→8	sub→and	**sub**	**lw**			
↑ 3	8→8	and	**sub**	**nop**	**lw**		

如果没有转发电路呢？

- 由于有转发电路，因此lw指令只插入1个NOP指令
- Q：如果没有转发，需要怎么处理呢？
- A：EX/MEM，MEM/WB也均需要做冲突分析及NOP处理
  - EX/MEM，MEM/WB也需要修改，并增加相应控制信号

地址	指令
0	lw $t0, 0($t1)
4	sub $t3, $t0, $t2
8	and $t5, $t0, $t4
12	or $t7, $t0, $t6
16	add $t1, $t2, $t3

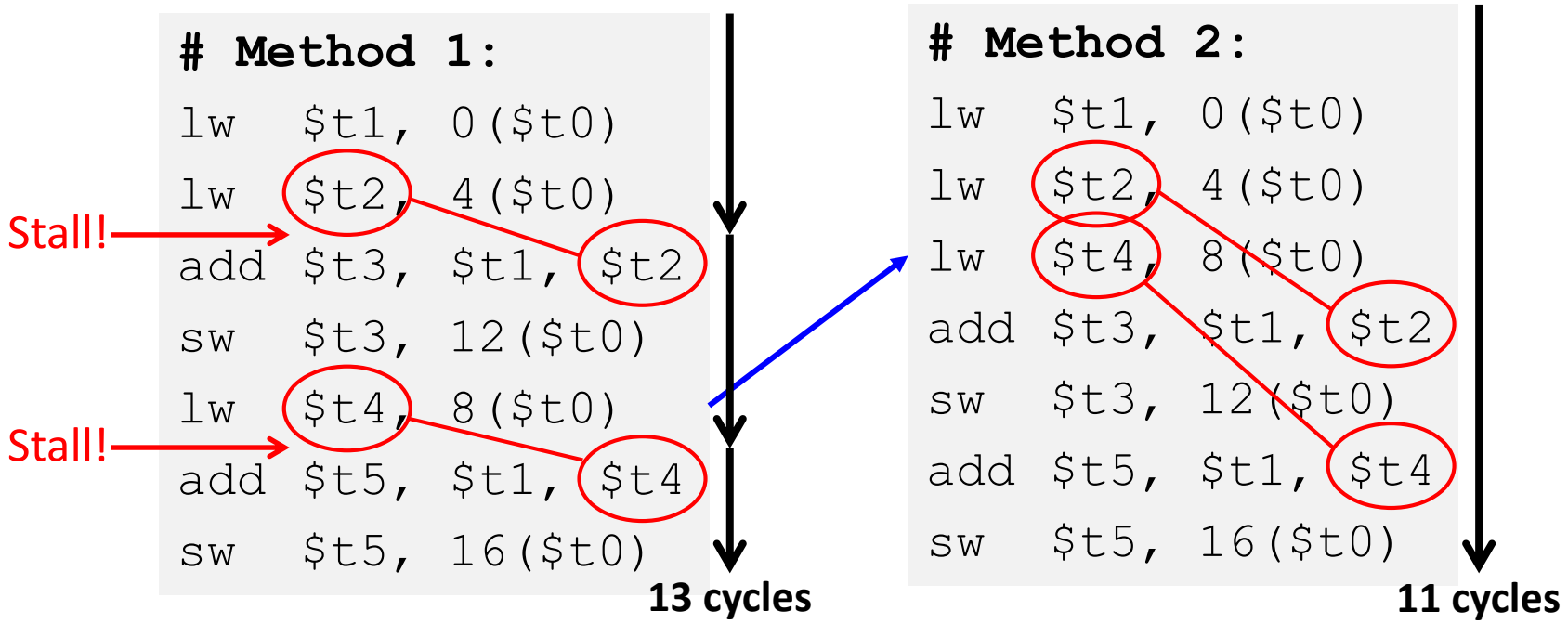
				RF(读)	ALU	DM		
CLK	PC	IM	IF/ID	ID/EX	EX/MEM	MEM/WB	RF
↑ 1	0→4	lw→sub	**lw**				
↑ 2	4→8	sub→and	**sub**	**lw**			
↑ 3	8	and	**sub**	**nop**	**lw**		
↑ 4	8	and	**sub**	**nop**	**nop**	**lw结果**	
↑ 5	8	and	**sub**	**nop**	**nop**	**nop**	**lw结果**
↑ 6	8→12	and→or	**and**	**sub**	**nop**	**nop**	**nop**

Data Hazard: Loads (4/4)

- Slot after a load is called a *load delay slot*
  - If that instruction uses the result of the load, then the hardware interlock will stall it for one cycle
  - Letting the hardware stall the instruction in the delay slot is equivalent to putting a `nop` in the slot (except the latter uses more code space)
- **Idea:** Let the compiler put an unrelated instruction in that slot → no stall!

Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction!
- MIPS code for `A=B+E; C=B+F;`



Agenda

- Structural Hazards
- Data Hazards
  - Forwarding
- Data Hazards (Continued)
  - Load Delay Slot
- Control Hazards
  - Branch and Jump Delay Slots
  - ~~Branch Prediction~~

3. Control Hazards

- Branch (`beq`, `bne`) determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction
    - Still working on ID stage of branch
- **Simple Solution:** Stall on *every* branch until we have the new PC value
  - How long must we stall?

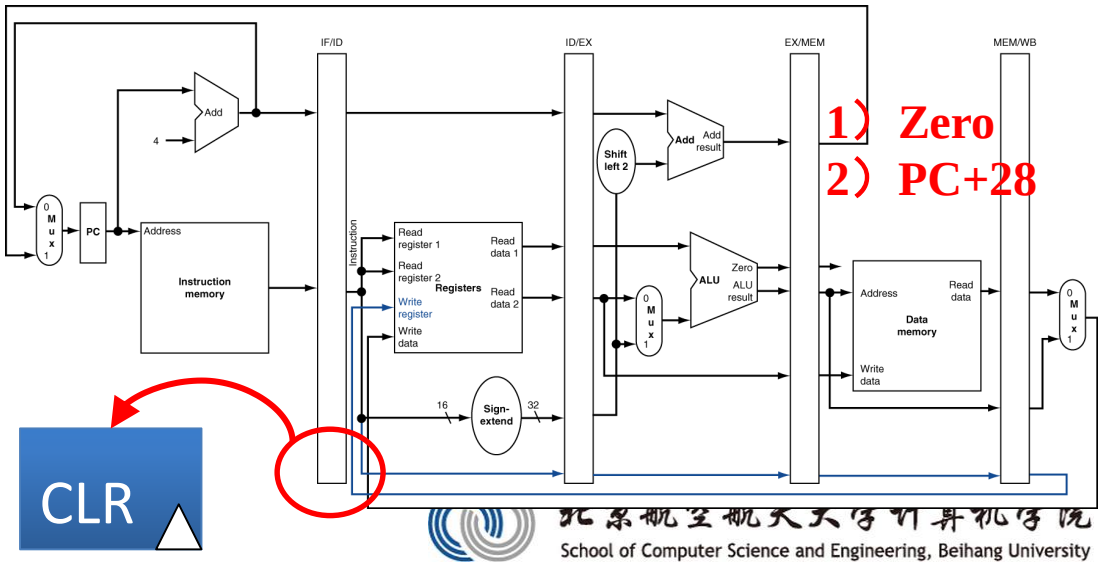
B指令冒险造成的停顿代价

地址	指令
0	beq $1, $3, 24
4	and $12, $2, $5
8	or $13, $6, $2
12	add $14, $2, $2
28	lw $4, 100($7)

	NPC		RF(读)	ALU	DM		
CLK	PC	IM	D	E	M	W	RF
↴1	0	beq	beq				
	4	and					
↴2	4	and	nop(1)	beq			
	4	and					
↴3	4	and	nop(2)	nop(1)	beq		
	4	and			Zero 1		
↴4	4	and	nop(3)	nop(2)	nop(1)		
	28	lw					
↴5	28	lw	lw	nop(3)	nop(2)	nop(1)	
	32	XXX					

- 如不对B指令做任何处理，则必须插入3个NOP
  - b指令结果及新PC值保存在EX/MEM，因此PC在clk4才能加载正确值
  - IF/ID在clk5才能存入转移后指令(即lw指令)

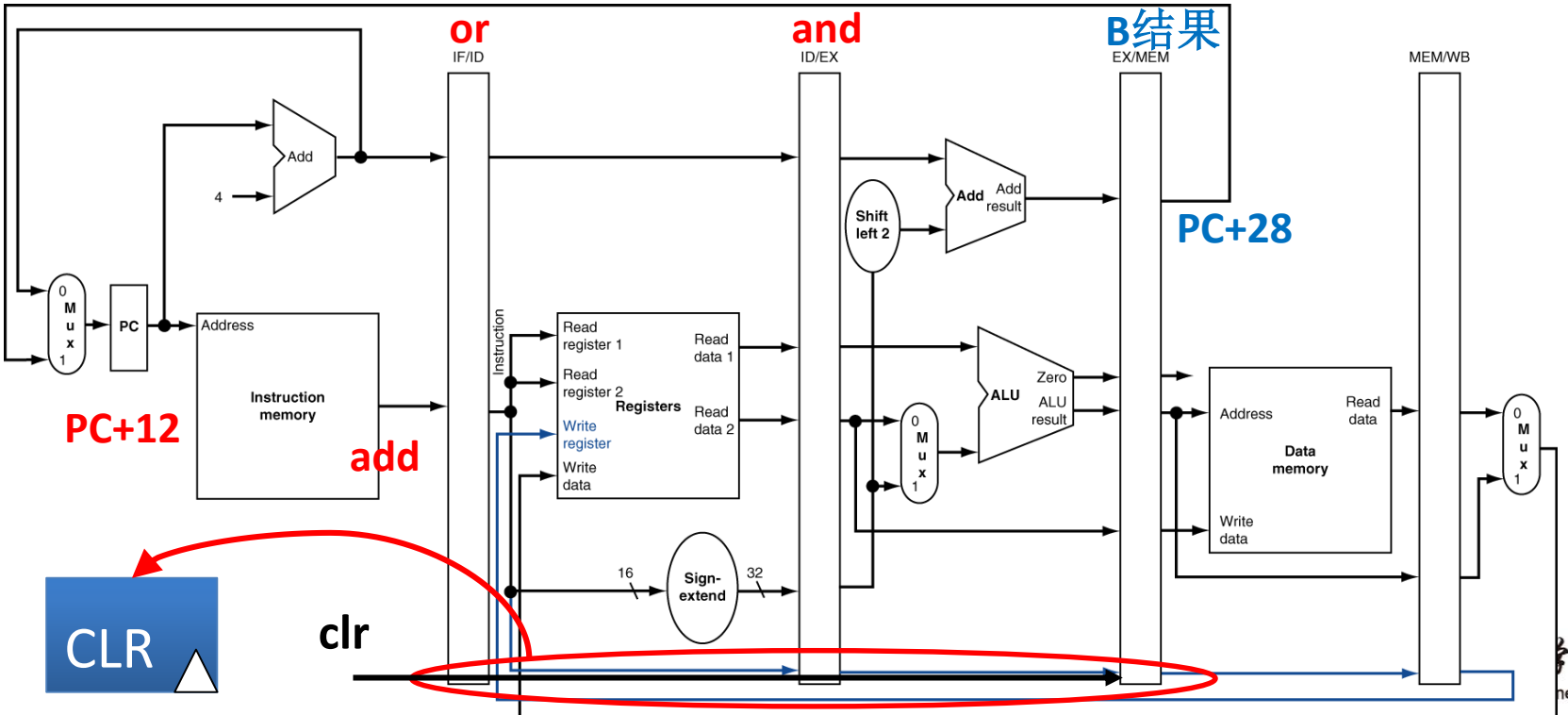
Q：clr的表达式？



方案1：假定分支不发生

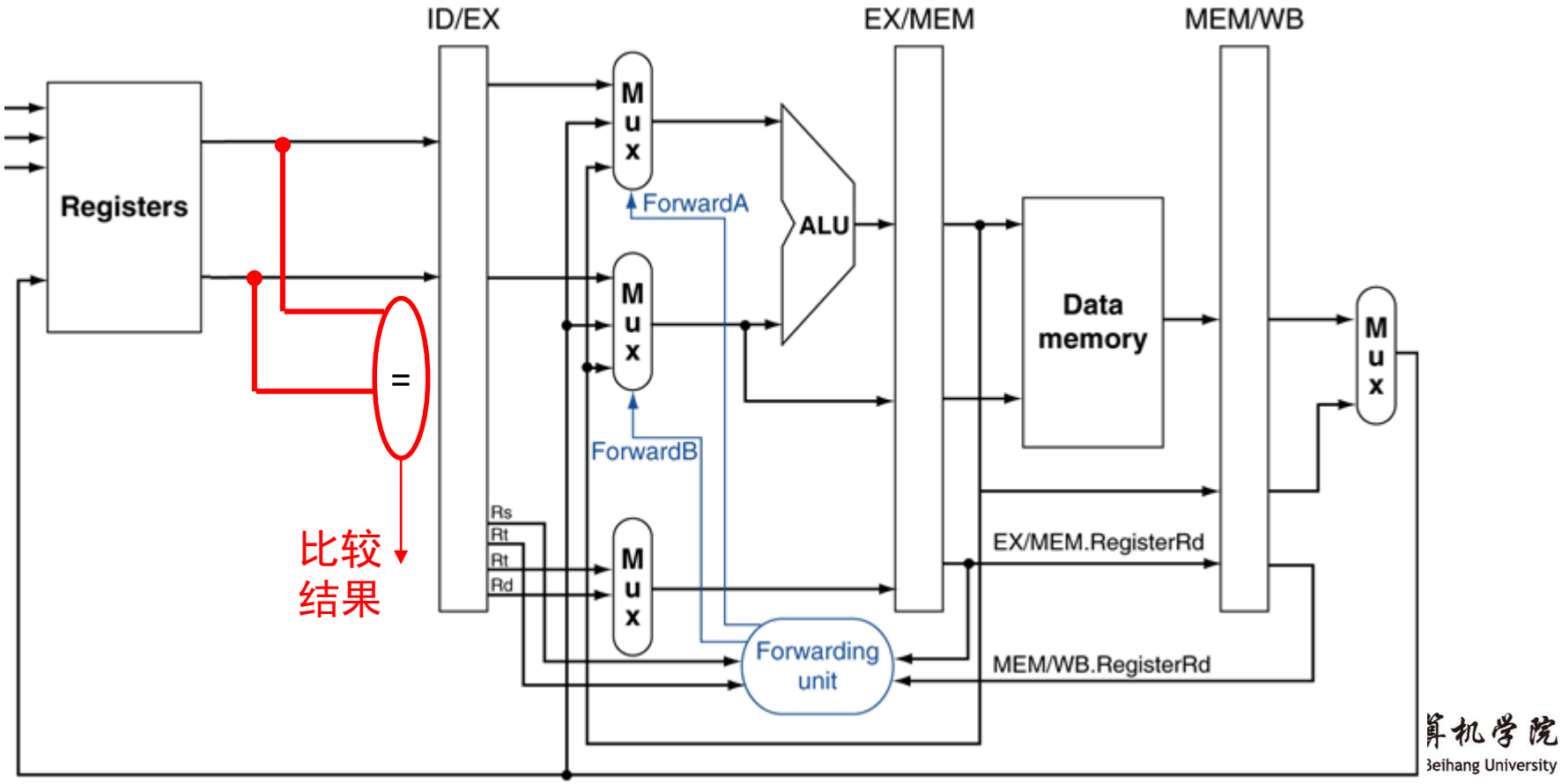
- 即使在ID级发现是B指令也不停顿
- 根据B指令结果，决定是否清除3条后继指令
  - 使得and/or/add不能前进

PC相对偏移	指令
0	beq $1, $3, 24
4	and $12, $2, $5
8	or $13, $6, $2
12	add $14, $2, $2
28	lw $4, 50($7)



方案2：缩短分支延迟

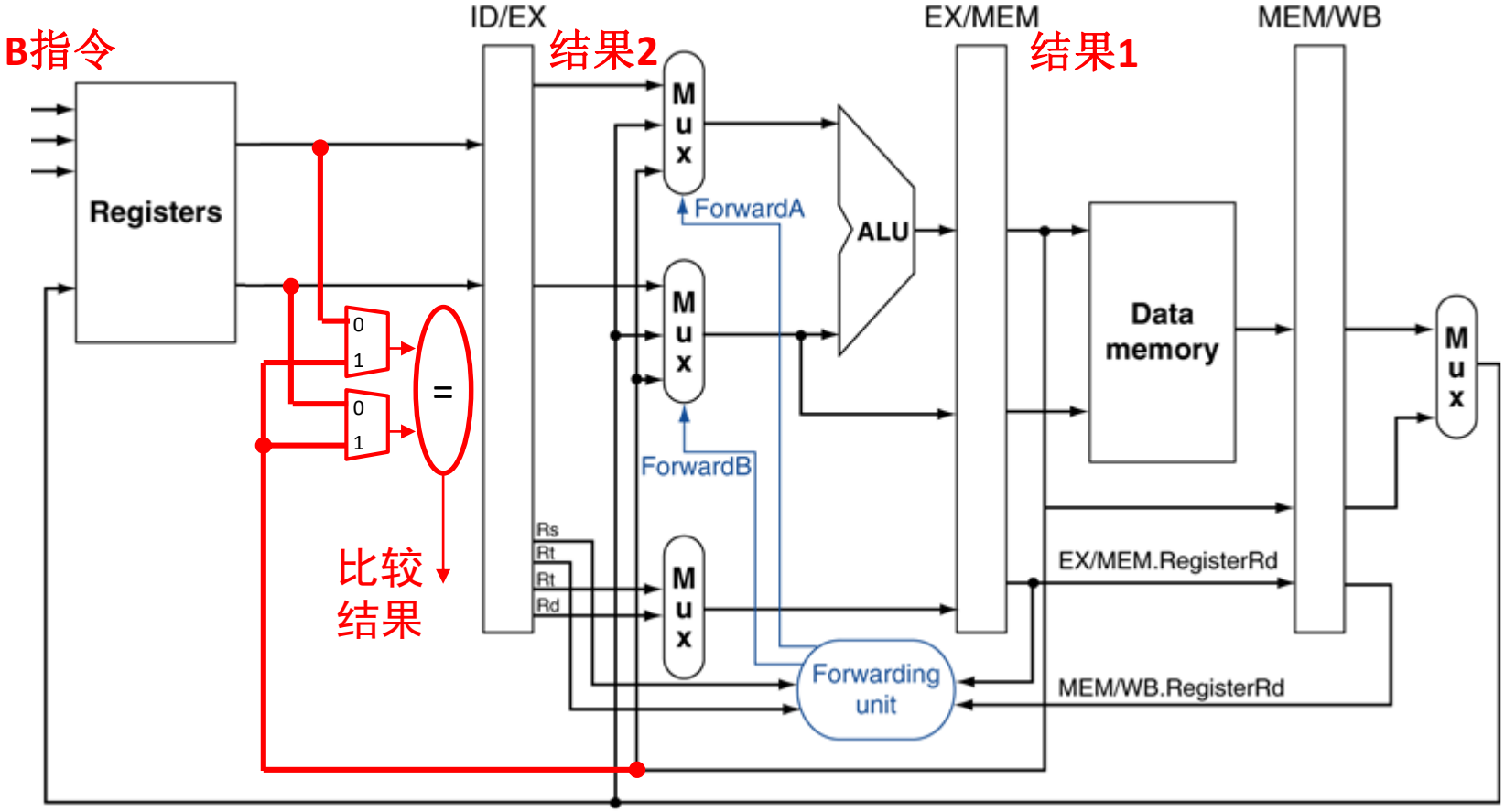
- 在ID阶段放置比较器，尽快得到B指令结果
  - B指令结果可以提前2个clock得到
  - B指令后继可能被废弃的指令减少为1条
    - 当需要转移时，清除IF/ID即可



方案2：缩短分支延迟

- 比较器前置后，会产生数据相关
  - B指令可能依赖于前条指令的结果
- 依赖结果1：从ALU转发数据
- 依赖结果2：只能暂停

Q：如果依赖MEM/WB的结果，是否需要设置转发？
A：转发！



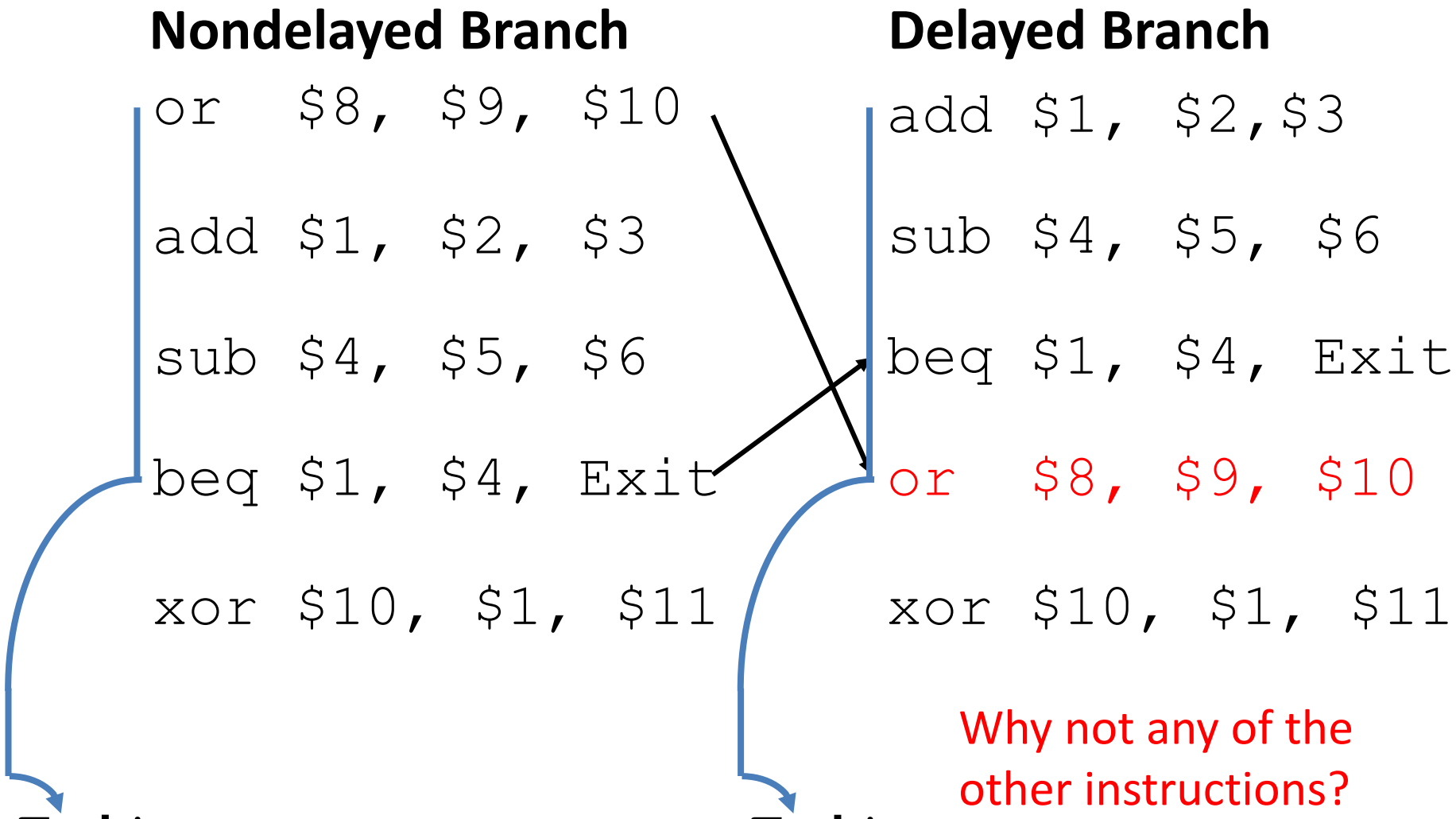
3. Control Hazard: Branching

- **Option #3:** *Branch delay slot*
  - Whether or not we take the branch, *always* execute the instruction immediately following the branch
  - Worst-Case: Put a `nop` in the branch-delay slot
  - Better Case: Move an instruction from before the branch into the branch-delay slot
    - Must not affect the logic of program

3. Control Hazard: Branching

- MIPS uses this *delayed branch* concept
  - Re-ordering instructions is a common way to speed up programs
  - Compiler finds an instruction to put in the branch delay slot ≈ 50% of the time
- Jumps also have a delay slot
  - Why is one needed?

Delayed Branch Example



Delayed Jump in MIPS

- MIPS Green Sheet for `jal`:
  `R[31]=PC+8; PC=JumpAddr`
  - `PC+8` because of *jump delay slot*!
  - Instruction at `PC+4` always gets executed before `jal` jumps to label, so return to `PC+8`

Summary

- Hazards reduce effectiveness of pipelining
  - Cause stalls/bubbles
- Structural Hazards
  - Conflict in use of datapath component
- Data Hazards
  - Need to wait for result of a previous instruction
- Control Hazards
  - Address of next instruction uncertain/unknown
  - Branch and jump delay slots